A note on an L-approach for solving the manufacturer's pallet loading problem

Ernesto G. Birgin [*] Reinaldo Morabito [†] Fabio H. Nishihara [‡]

November 19th, 2004.

Abstract

An L-approach for packing (l, w)-rectangles into an (L, W)-rectangle was introduced in an earlier work by Lins, Lins and Morabito. They conjecture that the L-approach is exact and point out its runtime requirements as the main drawback. In this note it is shown that, by simply using a different data structure, the runtime is considerably reduced in spite of larger (but affordable) memory requirements. This reduction is important for practical purposes since it makes the algorithm much more acceptable for supporting actual decisions in pallet loading. Intensive numerical experiments showing the efficiency and effectiveness of the algorithm are presented.

Key words: Cutting and packing, pallet and container loading, recursive algorithm, implementation.

Introduction

An interesting case of cutting and packing problems is loading products (packaged in boxes) on a rectangular pallet in such a way as to optimize pallet utilization. This problem is known as the manufacturer's pallet loading problem if all boxes are identical. This is the situation of a manufacturer that produces goods packaged in identical boxes of size (l, w, h), which are then arranged in horizontal layers on pallets of size (L, W, H) (where H is the maximum height of the loading). It is assumed that the boxes are available in large quantities and are orthogonally loaded on each pallet (that is, with their sides parallel to the pallet sides).

[*] Department of Computer Science, Institute of Mathematics and Statistics, University of São Paulo, Rua do Matão 1010, Cidade Universitária, 05508-090 São Paulo, SP - Brazil (egbirgin@ime.usp.br). Sponsored by FAPESP (Grants 01/04597-4, 02/00094-0 and 03/09196-6), CNPq (Grant 302266/2002-0) and Pronex.

[†] Department of Production Engineering, Federal University of São Carlos, Via Washington Luiz km. 235, 13565-905, São Carlos, SP - Brazil (morabito@power.ufscar.br). Sponsored by FAPESP (Grant 01/2972-2) and CNPq (Grant 522973/95-7)

[‡] Department of Computer Science, Institute of Mathematics and Statistics, University of São Paulo, Rua do Matão 1010, Cidade Universitária, 05508-090 São Paulo, SP - Brazil (fhn@ime.usp.br). Sponsored by FAPESP (Grant 03/00460-0)

If an orientation is fixed, the problem consists of packing the maximum number of (l, w)-rectangles and (w, l)-rectangles orthogonally into a larger rectangle without overlapping, yielding a layer of height h. Although there are polynomial-time algorithms for the guillotine version of this problem (Tarnowski *et al*[1]), the non-guillotine problem is widely claimed, but yet not proven, to be NP-complete (Dowsland[2], Nelissen[3]). Actually, the decision version of the problem is not known to be in NP at all (Nelissen[4], Letchford and Amaral[5]).

A number of authors have dealt with the manufacturer's pallet loading problem as discussed in Lins *et al*[6], Pureza and Morabito[7] and Alvarez-Valdes *et al*[8,9] (see also the references therein). In particular, in Lins *et al*[6], an L-approach based on a recursively defined function for packing (l, w)-rectangles into a larger rectangular (or L-shaped) piece was presented. Such approach is able to optimally solve all testing problems (more than 20,000 representatives of infinite equivalence classes of the literature), including the 16 hard instances unresolved by other heuristics. Such testing problems cover all instances with solutions of up to 100 (l, w)-rectangles and pallet dimension (L, W) of up to $(1000, 1000)$ from problem sets Cover I and II in Morabito and Morales[10]. These problems are realistic in pallet loading contexts as depicted in Morabito *et al*[11].

Based on those results, Lins *et al*[6] conjectured that the L-approach always finds optimal packings of (l, w)-rectangles into an (L, W)-rectangle. Nevertheless, the main drawback of the approach is its requirement in terms of computer runtime. For example, there are instances for which the approach takes hundreds of minutes (on a 700Mhz Pentium III processor) to solve them. In this note, we show that by simply using a different data structure in a C language implementation of the algorithm, the runtimes can be considerably reduced (from hundreds of minutes to a few minutes in hard instances), despite larger (but affordable) memory requirements. This reduction is important for practical purposes since it makes the algorithm much more acceptable for supporting actual decisions in pallet loading.

This note is organized as follows: in the next section we discuss some details of the algorithm implementation. Then we present the numerical experiments. Finally, the last section contains some concluding remarks.

Implementation of the algorithm

The L-approach is based on the computation of a recursive formula of dynamic programming that deals with a huge number of subproblems. Given that the same subproblem may appear several times along the recursion, and to be able to build up the solution at the end, it is fundamental to keep the information of the subproblems previously solved in memory. This information basically relates to the way each subproblem was solved and the number of packed rectangles in its solution. Therefore, an important aspect of the L-approach implementation is how to store and retrieve information of each solved subproblem. Note that there is a trade-off between the amount of memory required and rapid information access.

Each subproblem is related to an L-shaped piece in which the (l, w)-rectangles must be packed. A *standardly positioned* L-shaped piece[6], represented by (X, Y, x, y) where $x \le X$ and $y \le Y$, is defined as the topological closure of the rectangle whose diagonal goes from $(0, 0)$ to (X, Y) minus the rectangle whose diagonal goes from (x, y) to (X, Y), as shown in Figure 1. Assume that L, W, l, w are non-negative integers such that $L \ge W$ and $l \ge w$, and consider the increasing and finite sequence $Z = z_0, z_1, \dots, z_p$ of all the non-negative integer combinations z_i of l and w, such that $z_i \le L$. All subproblems generated by the algorithm correspond to standardly positioned (X, Y, x, y) such that $X, Y, x, y \in Z$. Hence, given a subproblem (X, Y, x, y), there are unique indices I,J,i and j satisfying $X = z_I$, $Y = z_J$, $x = z_i$ and $y = z_j$. In other words, for each subproblem (X, Y, x, y) there is a unique 4-uple (I, J, i, j).

Figure 1 about here.

In Lins *et al*[6] a data structure named *phorma* (Lins *et al*[12]) was used to accomplish an efficient indexation of (I, J, i, j). Phorma (acronym for perfectly hashable order restricted multidimensional array) is a data structure for perfect hashing of multidimensional arrays which have order restrictions on their entries. This is the case in the L-approach, where we would like to enumerate the quatruples of non-negative numbers $(X, Y, x, y) \le (L, W, L, W)$ that satisfy

$$B_L = (X \ge x) \wedge (Y \ge y) \wedge (X \ge Y) \wedge ((X \ne Y) \vee (x \ge y)) \wedge$$
$$((X \ne x) \vee (Y = y)) \wedge ((Y \ne y) \vee (X = x)).$$

The solution adopted in phorma is based on the theory of combinatorial families developed in Nijenhuis and Wilf[13]. The central idea is to associate a digraph to a collection of combinatorial objects in such a way that each object in the family is in $1-1$ correspondence with a path in the digraph. Under mild assumptions, the digraph is logarithmically smaller than the number of objects to be enumerated. Moreover, its construction requires the enumeration and lexicographical ordering of a reduced set of objects related to (but much smaller than) the original set of objects to be enumerated. Finally, the perfect hash function can be computed at the expense of computing a path in the digraph. Moreover, many savings can be done in the construction and storage of the related data structures, doing most of the tasks at compilation time. See Lins *et al*[12] for details (including several examples for typical values of some phorma parameters).

Phorma has a good compromise between memory requirements and access time. However, for the problem sizes treated in Lins *et al*[6], it is possible to apply a simpler data structure that requires much more memory but substantially reduces the main drawback of the L-approach implementation presented in Lins *et al*[6]: its computer runtime.

Let $\alpha(\cdot) : Z \to \{0, 1, \dots, p\}$, $\alpha(z_i) = i$, and $\alpha^{-1}(\cdot) : \{0, 1, \dots, p\} \to Z$, $\alpha^{-1}(i) = z_i$, for $i = 0, 1, \dots, p$, be a bijective function and its inverse to perform the indexation of the subproblems. Consider two auxiliary vectors u and v with sizes $L + 1$ and $p + 1$, respectively, such that

$$u[k] = \begin{cases} i, & \text{if there exists } i \text{ such that } k = z_i \in Z, \\ \text{undefined}, & \text{otherwise,} \end{cases}$$

for $k = 0, 1, \dots, L$, and $v[k] = z_k, k = 0, 1, \dots, p$. Then, $\alpha(z_i) = u[i]$ e $\alpha^{-1}(i) = v[i]$. That is, both functions can be evaluated in constant time. Figure 2 shows vectors u and v for a problem instance with $L = 10$, $W = 4$, $l = 5$ e $w = 2$.

Figure 2 about here.

It should be noted that, for a subproblem (X, Y, x, y), the indices

$$(I, J, i, j) = (\alpha(X), \alpha(Y), \alpha(x), \alpha(y))$$

such that $X = z_I$, $Y = z_J$, $x = z_i$ and $y = z_j$ are computed in constant time. In this way, the information of each subproblem can be saved at position (I, J, i, j) of a 4-dimensional array (with each dimension of size $p+1$). Observe that the data access is direct (constant computational cost) at the expense of many allocated but non-utilized array positions. The memory requirement is $O(p^4)$.

Numerical experiments

We implemented the L-approach using function α and its inverse for the indexation of the subproblems. The algorithm was coded in C language and the experiments run on an 1533 MHz AMD Athlon (TM) MP 1800+ processor, 512 Mb of RAM memory and Linux operating system. An approximate comparison (*http://www.spec.org*) indicates that this computer is less than twice as fast as the one used in Lins *et al*[6] (700 MHz Pentium III processor). The compiler was gcc version 2.95.4 with -O4 flag to optimize the code.

Table 1 shows the results obtained for the 16 problems analyzed in Lins *et al*[6]; these problems are hard to solve for block heuristics as shown in Morabito and Morales[10]. In the table the columns show the problem $P_n(L, W, l, w)$ (which means that n (l, w)-rectangles were packed into a pallet of dimensions (L, W)), the total amount of memory used by the 4-dimensional array (in MegaBytes) and the percentage actually used (to store the information of the subproblems), and the CPU runtime in seconds. Note that, on average, 1.52% of the elements of the 4-dimensional array is actually used by the algorithm.

The amounts of memory displayed in Table 1 are the ones used for the array utilized to save the subproblem information. The total amount of physical memory used by the task (including the size of the task's code, data and stack) is at most twice of the reported amounts. So, any computer with 512Mb of RAM memory is capable of solving these problems.

Table 1 about here.

In addition to the 16 instances presented in Table 1, we also solved more than 20,000 problems selected from sets Cover I and II in Morabito and Morales[10]. These instances correspond to all problems satisfying $L, W \leq 1000$. Table 2 shows the average, the standard deviation, the minimum and the maximum runtimes (in seconds) for each set of selected problems.

Table 2 about here.

Note that the average runtimes are affordable (with relatively high standard deviations). As pointed out in Lins *et al*[6], the L-approach solved all problems to optimality. This fact reinforces the conjecture of the algorithm exactness. The problems in Cover I (with up to 50 boxes) are on average much easier to solve, whereas the ones in Cover II (51 to 100 boxes) are on average as difficult as the 16 problems. Since most problems in Covers I and II are quickly solved by the block heuristic in Morabito and Morales[10], the runtimes of Table 2 can be substantially reduced combining this heuristic to the L-approach.

Concluding remarks

In this note we show that the L-approach introduced in Lins *et al*[6] can be effective for solving pallet loading problems of realistic sizes. By simply using a different data structure, the runtimes are considerably reduced in spite of larger (but affordable) memory requirements. This way, the computer requirements of the algorithm become more acceptable and the L-approach is a real option for pallet loading in logistics environments.

The present implementation (including the source code in C language) is available for benchmark purposes[14].

Acknowledgement: The authors thank the anonymous referee for his/her useful comments.

References

[1] A. Tarnowski, J. Terno and G. Scheithauer, A polynomial-time algorithm for the guillotine pallet loading problem, *INFOR* 32, pp. 275–287, 1994.

[2] K. A. Dowsland, An exact algorithm for the pallet loading problem, *European Journal of Operational Research* 84, pp. 78–84, 1987.

[3] J. Nelissen, How to use the structural constraints to compute an upper bound for the pallet loading problem, *European Journal of Operational Research* 84, pp. 662–680, 1995.

[4] J. Nelissen, Solving the pallet loading problem more efficiently, *working paper*, Graduiertenkolleg Informatik und Technik, Aachen, 1994.

[5] A. Letchford and A. Amaral, Analysis of upper bounds for the pallet loading problem, *European Journal of Operational Research* 132, pp. 582–593, 2001.

[6] L. Lins, S. Lins and R. Morabito, An L-approach for packing (l, w)-rectangles into rectangular and L-shaped pieces, *Journal of the Operational Research Society* 54, pp. 777–789, 2003.

[7] V. Pureza and R. Morabito, Some experiments with a simple tabu search algorithm for the manufacturer's pallet loading problem, *Computers & Operations Research*, to appear.

[8] R. Alvarez-Valdes, F. Parreño and J. M. Tamarit, A branch-and-cut algorithm for the pallet loading problem, *Computers & Operations Research*, to appear.

[9] R. Alvarez-Valdes, F. Parreño and J. M. Tamarit, A tabu search algorithm for the pallet loading problem, *OR Spectrum* 27, pp. 43–61, 2005.

[10] R. Morabito and S. Morales, A simple and effective recursive procedure for the manufacturer's pallet loading problem, *Journal of the Operational Research Society* 49, pp. 819–828, 1998.

[11] R. Morabito, S. Morales and J. Widmer, Loading optimization of palletized products on trucks, *Transportation Research (Part E)* 36, pp. 285–296, 2000.

[12] L. Lins, S. Lins and S. Melo, Phorma: perfectly hashable order restricted multidimensional arrays, *Discrete Applied Mathematics* 141, pp.209–223, 2004.

[13] A. Nijenhuis and H. S. Wilf, Combinatorial algorithms for computers and calculators, Academic Press (second edition), 1978.

[14] www.ime.usp.br/~egbirgin.

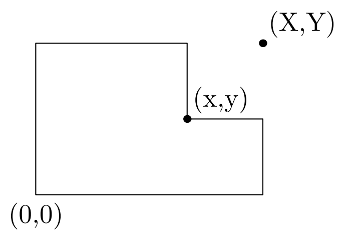

Figure 1: Standardly positioned L-shaped piece represented by (X, Y, x, y).

u	0		1		2	3	4	5	6	7	8
	0	1	2	3	4	5	6	7	8	9	10

v	0	2	4	5	6	7	8	9	10
	0	1	2	3	4	5	6	7	8

Figure 2: Vectors u and v for a problem instance with $L = 10$, $W = 4$, $l = 5$ e $w = 2$.

Problem	4-dimensional array memory requirement (in Megabytes)		Runtime
	Allocated	Actually used	(in seconds)
$P_{53}(43, 26, 7, 3)$	2.98	0.981 %	2.48
$P_{57}(49, 28, 8, 3)$	4.89	1.375 %	7.62
$P'_{69}(57, 34, 7, 4)$	8.25	1.242 %	13.25
$P''_{69}(63, 44, 8, 5)$	8.94	1.215 %	15.18
$P_{71}(61, 35, 10, 3)$	11.29	1.821 %	37.52
$P_{75}(67, 37, 11, 3)$	16.19	1.858 %	68.80
$P'_{77}(61, 38, 10, 3)$	11.29	2.392 %	56.35
$P''_{77}(61, 38, 6, 5)$	10.46	1.237 %	19.45
$P_{81}(67, 40, 11, 3)$	16.19	2.434 %	103.03
$P'_{82}(74, 49, 11, 4)$	18.54	2.750 %	134.88
$P''_{82}(93, 46, 13, 4)$	47.72	1.298 %	168.38
$P'_{96}(106, 59, 13, 5)$	67.89	1.463 %	327.88
$P''_{96}(141, 71, 13, 8)$	143.05	0.396 %	141.23
$P_{97}(74, 46, 7, 5)$	22.53	1.197 %	57.73
$P_{99}(86, 52, 9, 5)$	36.35	1.852 %	191.28
$P_{100}(108, 65, 10, 7)$	64.68	0.732 %	111.87
Average	30.70	1.52 %	91.06

Table 1: Results obtained for the 16 hard problems.

Data set	# selected problems	Runtimes (in seconds)			
		Average	Standard deviation	Min	Max
Problems in Lins *et al*[6]	16	91.06	87.39	2.48	327.88
Cover I	3,179	1.10	3.19	0.00	50.20
Cover II	16,938	113.10	217.55	0.00	3096.45

Table 2: Runtime statistics for the more than 20,000 problems selected from data sets Covers I and II.