TSpack: A Unified Tabu Search Code for Multi-Dimensional Bin Packing Problems *

Andrea Lodi, Silvano Martello, Daniele Vigo

D.E.I.S., University of Bologna
Viale Risorgimento, 2 - 40136 - Bologna (Italy)
E-mail: {alodi, smartello, dvigo}@deis.unibo.it

We present a computer code that implements a general Tabu Search technique for the solution of two- and three-dimensional bin packing problems, as well as virtually any of their variants requiring the minimization of the number of bins. The user is only requested to provide a procedure that gives an approximate solution to the actual variant to be solved.
Keywords: Packing, Cutting, Tabu Search

1. Introduction

Several real-world optimization problems in the cutting and packing area require to allocate, without overlapping, a set of rectangular *items* to larger identical rectangular standardized stock units (often called *bins*) by minimizing the number of needed bins.

In the *Two-Dimensional Bin Packing Problem* (2BP), each item j ($j = 1, \ldots, n$) is defined by its *width*, w_j, and *height*, h_j, and the bins are rectangles of width W and height H. The problem arises, e.g., in wood and glass industries (cutting of rectangular components from large sheets of material), in warehousing contexts (placement of goods on shelves), in newspapers paging (arrangement of articles and advertisements into pages). In practical applications, a number of variants arise. The items may either have a fixed orientation, or it may be admissible to rotate them in order to obtain a better packing. In cutting contexts it may be required that the produced patterns are *guillotine cuttable*, i.e., such

* Support given by the Ministero dell'Istruzione, dell'Università e della Ricerca (MIUR) and the Consiglio Nazionale delle Ricerche (CNR), Italy.

that the items are obtained through a sequence of edge-to-edge cuts parallel to the edges of the bin. Research on 2BP started in the Eighties: Chung, Garey and Johnson [4] and Frenk and Galambos [8] studied approximation algorithms with asymptotic worst-case performance guarantee, Berkey and Wang [2] presented extensions of classical one-dimensional bin packing approximation algorithms, and Bengtsson [1] proposed application-oriented heuristics. More recently, Martello and Vigo [18] analyzed lower bounds and presented an exact branch-and-bound approach, while Lodi, Martello and Vigo [11–13] proposed heuristic algorithms, and developed the Tabu Search approach whose implementation is discussed here.

In the *Three-Dimensional Bin Packing Problem* (3BP), the items are boxes of width w_j, height h_j and *depth* d_j, while the bins are defined by width W, height H and depth D. The problem finds applications in the loading area (containers, trucks, ...) and in cutting contexts. In this case too, variants may consider item rotation and guillotine cutting. In addition, robot packable patterns have industrial relevance: a *robot packing* can be achieved by successively placing the items starting from the bottom-left-behind corner, and is such that each item is in front of, right of, or above each of the previously placed items. A heuristic algorithm for 3BP was presented by Scheithauer [20]. Chen, Lee and Shen [3] gave an integer programming formulation for the case where the bins may have different sizes. Martello, Pisinger and Vigo [17] proposed an exact algorithm (see also Pisinger, den Boef, Korst, Martello and Vigo [19]), while a Tabu Search approach was presented by Lodi, Martello and Vigo [14].

Surveys on packing problems have been given by Dyckhoff and Finke [6], Dowsland and Dowsland [5], Lodi, Martello and Vigo [15] and Lodi, Martello and Monaci [10], while Dyckhoff, Scheithauer and Terno [7] have presented an annotated bibliography.

Both 2BP and 3BP generalize the well-known *One-Dimensional Bin Packing Problem* (1BP), hence they are strongly NP-hard.

Computational experiments show that the exact solution of 2BP and 3BP instances may be attained only for moderate-size instances, while the use of heuristics is needed for larger instances. The object of this paper is the presentation of a computer code, TSpack, that implements the general Tabu Search technique developed in [13,14]. The code can be used for the solution of both 2BP and 3BP, and virtually any of their variants calling for the minimization of the number of used bins (including those allowing item rotation or requiring guillotine cutting). The user is only requested to provide a procedure that gives an approximate solution to the actual variant to be solved.

The algorithm is described Section 2, and the computer code is illustrated in Section 3. Extensions to other packing problems are discussed in Section 4.

2. The algorithm

The general structure of algorithm TSpack is given in Figure 1, adapted from [13].

algorithm TSpack:
  $z^* := A(\{1, \ldots, n\})$ (**comment:** incumbent solution value);
  let L be a lower bound on the optimal solution value;
  if $z^* = L$ **then stop**;
  initialize all tabu lists to empty;
  pack each item into a separate bin;
  $z := n$ (**comment:** Tabu Search solution value);
  $d := 1$;
  determine the target bin t;
  while time limit is not reached **do**
    diversify := `false`; $k := 1$;
    while *diversify* = `false` **and** $z^* > L$ **do**
      $k_{in} := k$;
      call SEARCH(t,k,*diversify*,z);
      $z^* := \min\{z^*, z\}$;
      if $k \leq k_{in}$ **then** determine the new target bin t
    end while;
    if $z^* = L$ **then stop**
    else call DIVERSIFICATION(d,z,t)
  end while
end.

Figure 1. Algorithm TSpack.

The main characteristic of the approach is a unified parametric neighborhood which is independent of the specific problem considered, and whose size is dynamically varied during the search. **More precisely, at each iteration of the main loop the size and the structure of a neighborhood are de-**

termined and a specific procedure, called SEARCH, to explore it is invoked. According to the output of procedure SEARCH (given in Figure 2 and discussed later in detail), the method is iterated until a time limit is reached. There are a tabu list and a tabu tenure for each neighborhood size.

The specific constraints of the problem just appear in a deterministic inner heuristic, A, that produces a feasible solution for a given instance (or sub-instance) of 2BP or 3BP. Algorithm A is used to evaluate the moves within the neighborhood search. Let $A(S)$ be the output solution value returned by A when invoked for a (sub-)instance of the problem induced by an input item set S.

Experimental observations show that the objective functions of multi-dimensional bin packing problems are "flat", in the sense that very many different solutions use the same number of bins. Algorithm TSpack introduces a characterization of equivalent solutions, based on the existence of a bin that packs less and/or smaller items than the others, hence is more likely to be emptied through local optimization. The *moves* try to empty this *target bin*, t, by changing the packing of a subset S of items made up by one item, say j, from bin t, plus the current contents of k other bins, where k defines the current neighborhood size. A new solution is then obtained by adding the $A(S)$ bins produced by A to the bins that currently pack items $\{1, \ldots, n\} \setminus S$. **This solution is considered "feasible" if the overall number of bins used do not exceed the one of the current solution and, obviously, the packing is changed. In other words, we are looking for moving item j outside from bin t without creating extra bins.**

The target bin is determined as follows. Let S_i denote the set of items currently packed into bin i, and α a user-specified positive value. Bin t is then the one minimizing, over all current bins i, the *filling function*

$$\varphi(S_i) = \alpha \frac{\sum_{j \in S_i} v_j}{V} - \frac{|S_i|}{n} \tag{1}$$

where

$$\begin{cases} v_j = w_j h_j & \text{and } V = WH \quad \text{for } 2\text{BP} \\ v_j = w_j h_j d_j & \text{and } V = WHD \quad \text{for } 3\text{BP} \end{cases} \tag{2}$$

As mentioned, the neighborhood is searched by procedure SEARCH, given in Figure 2, taken from [13]. For each item j in bin t, algorithm A is executed on all the sub-instances induced by j and by all k-tuples of other bins. Parameter

```
procedure SEARCH(t,k,diversify,z):
    penalty* := +∞;
    for each j ∈ S_t do
        for each k-tuple K of bins not including t do
            S := {j} ∪ (⋃_{i∈K} S_i);
            penalty := +∞;
            case
                A(S) < k:
                    execute the move and update the solution value z;
                    k := max{1, k − 1};
                    return;
                A(S) = k:
                    if the move is not tabu or S_t ≡ {j} then
                        execute the move and update the solution value z;
                        if S_t ≡ {j} then k := max{1, k − 1};
                        return
                    end if;
                A(S) = k + 1 and k > 1:
                    let I be the set of k + 1 bins used by A;
                    t̄ := arg min_{i∈I}{φ(S_i)},  T := (S_t \ {j}) ∪ S_t̄;
                    if A(T) = 1 and the move is not tabu then
                        penalty := min{φ(T), min_{i∈I\{t̄}}{φ(S_i)}}
            end case;
            penalty* := min{penalty*, penalty};
        end for;
    end for;
    if penalty* ≠ +∞ then execute the move corresponding to penalty*
    else if k = k_max then diversify := true else k := k + 1
return.
```

Figure 2. Algorithm TSpack: procedure SEARCH.

k, that defines the size of the neighborhood, may be seen as a local intensification/diversification tool. Its value is updated as follows. When a move decreases the current number of used bins, or when a non-tabu move removes j from t by packing the sub-instance in exactly k bins, the move is immediately performed, the neighborhood size is reduced by one unit, and the control returns to the main algorithm. When, instead, the neighborhood has been completely searched without finding an acceptable move, its value is increased by one unit.

A move that is not immediately performed is evaluated through a *penalty*. The penalty is infinity if the move is tabu, or if algorithm A used at least two extra bins (i.e., $A(S) > k+1$), or if $k = 1$. Otherwise (i.e., $A(S) = k+1$ and $k > 1$), the penalty is computed as follows. We determine a local target bin $\bar{t}$ among the $k+1$ bins produced by A, and re-execute algorithm A on the sub-instance induced by the items in bin $\bar{t}$ plus the residual items in the target bin, in an attempt to get a single-bin solution. If this happens, the penalty of the overall move is the minimum among the filling function values computed for the $k+1$ resulting bins; otherwise, the move is not acceptable and its penalty is set to infinity.

When the neighborhood has been entirely searched without finding a move that has to be immediately performed, the acceptable move having the minimum penalty (if any) is performed and the control returns to TSpack. If, instead, no acceptable move has been found, the neighborhood is enlarged by increasing the current value of parameter k by one, or, if k already reached a maximum prefixed value $k_{\max}$, by executing a global diversification: according to the value of parameter d, two kinds of diversification are performed, as shown in Figure 3.

procedure DIVERSIFICATION(d,z,t):
    if $d \le z$ **and** $d < d_{\max}$ **then**
        $d := d + 1$;
        let t be the bin with d-th smallest value of $\varphi(\cdot)$;
    else
        remove from the solution the $\lfloor z/2 \rfloor$ bins with smallest $\varphi(\cdot)$ value;
        pack into a separate bin each item currently packed in a removed bin;
        reset all tabu lists to empty;
        $d := 1$
return.

Figure 3. Algorithm TSpack: procedure DIVERSIFICATION.

Each neighborhood has a tabu list and a tabu tenure τ_k ($k = 1, \ldots, k_{\max}$). For $k > 1$, each list stores the *penalty** values corresponding to the last τ_k moves performed in the corresponding neighborhood. For $k = 1$ instead, since no penalty is computed (see Figure 2), the tabu list stores the values of the filling function, $\varphi(\cdot)$, corresponding to the last τ_1 sets for which a move has been performed.

The key aspect of the framework described above is the switching among neighborhoods of different size. The main motivation of this resides in the attempt of interchanging efficiently *intensification* and *diversification* with the aim of emptying the target bin, thus improving the current solution. On one side, a small value of k implies, in general, the re-combination of only few items, somehow assuming that most of the current solution is "good". The corresponding neighborhoods are small and their exploration is fast. On the other side, greater values of k involve much more items, i.e. a wider neighborhood, possibly allowing a relevant change in the current solution.

3. The code

The ANSI-C code implementing algorithm TSpack is available at:

`http://www.or.deis.unibo.it/research_pages/ORcodes/ORcodes.htm`

and its use is free for academic purposes. The code was compiled using both `cc` and `gcc` compilers. It was also tested with the `-pedantic` option of `gcc` to check strict respect of the ANSI-C standard.

The n items are numbered from 0 to $n-1$. The position of the items in the solution is referred to a coordinate system having its origin in the lower-left(-front) corner of the bin.

A prototype of `TSpack` appears as

```
int TSpack(int d, int n, int **w, int *W, int lb, float TL,
           int *ub0, int **x, int *b)
```

Function `TSpack` returns the output number of used bins. The meaning of the input parameters is:

`d` number of dimensions, `d` ≥ 2 (tested values: 2, 3);

`n` size of the problem, i.e., number of items;

`w` `d`$\times$`n` array giving the item sizes, i.e., w, h (and d), where `w[0][j` $-$ `1]` $= w_j$, `w[1][j` $-$ `1]` $= h_j$ (and `w[2][j` $-$ `1]` $= d_j$) for $j = 1, \ldots, n$;

`W` `d`$\times$`1` array giving the bin sizes, i.e., W, H (and D), where `W[0]` $= W$, `W[1]` $= H$ (and `W[2]` $= D$);

`lb` lower bound on the optimal solution value (trivial value `lb` $= 1$ is acceptable);

`TL` time limit.

On output, the used bins are numbered from 0 to `TSpack`-1. The meaning of the output parameters is:

`ub0` intial upper bound as computed by the selected algorithm A;

`x` `d`$\times$`n` array giving the coordinates of the point where the item lower-left(-front) corner is packed, i.e., `x[0][j` $-$ `1]` $=$ w-coordinate of item j, `x[1][j` $-$ `1]` $=$ h-coordinate of item j (and `x[2][j` $-$ `1]` $=$ d-coordinate of item j) for $j = 1, \ldots, n$;

`b` `n`$\times$`1` array giving the number of the bin where each item is packed, i.e., item j is packed in bin `b[j-1]`$+1$.

Procedure `TSpack` implements the pseudo-code given in Figure 1, and invokes the procedure `SEARCH` implementing the inner loop described by the pseudo-code given in Figure 2. Both procedures (together with some utility functions) are included in the source-code file `TSpack.c`. The source-code file `driver.c` contains a simple driver program which reads from an input file either a 2BP or a 3BP instance and invokes procedure `TSpack`. A couple of simple 2BP and 3BP instances with $n = 10$ are also given to allow the user to initially test the code.

The current release of the code contains a pair of very simple heuristic algorithms which can be used as inner heuristic (algorithm A) to test Two- and Three-Dimensional problems. Specifically, we have implemented the classical *Hybrid Next Fit* algorithm for 2BP (see Johnson [9]), and a possible adaptation of the same algorithm in the 3BP context. Moreover, the code contains a straightforward implementation of the trivial *continuous* lower bound $LB = \lceil \sum_{j=1}^{n} v_j / V \rceil$, where v_j and V are defined as in (2).

In order to facilitate the user in experimenting specific approaches or in adapting the code to other packing contexts, procedures `TSpack` and `SEARCH` invoke generic procedures for:

(i) *inner heuristic*;

(ii) *lower bounding*;

(iii) *filling function*;

(iv) *penalty function*.

The user selects the algorithms by setting the corresponding parameters in the header file `TSpack.h`. In this way, he/she can easily enlarge the arsenal of the current release by adding new algorithms for the procedures above, and try several configurations by just changing the parameters' selection.

Finally, the same header file also contains a default setting of the parameters which characterize algorithm TSpack as described in Section 2.

Distribution package

In conclusion, the distribution package `TSpack.tar.gz` contains the files:

1. `TSpack.c` and `TSpack.h` (source and header codes);
2. `driver.c` (driver program);
3. `2d.in` and `3d.in` (2BP and 3BP sample instances).

Web Page and Benchmark

The web page indicated above contains, together with the distribution package, the output files of the algorithm on the classical Two- and Three-Dimensional benchmark instances described in [18] and [17], respectively. The source codes to generate these instances can be downloaded from the web sites:

`http://www.or.deis.unibo.it/research_pages/ORinstances/ORinstances.htm`

and

`http://www.diku.dk/~pisinger/codes.html`

for 2BP and 3BP, respectively. **Specifically, files ... and ... contain the results obtained by executing `TSpack` on 500 2BP and 320 3BP instances, one line for each test problem. The inner heuristics used for these tests are the ones provided with the distribution package and**

the execution is halted after 500 Tabu Search iterations. Thus, these results are just intended to give the user a starting point for his/her research. However, the results immediately show the good improvement obtained with respect to the simple inner heuristic. This is somehow obvious, but at the same time it is a very important feature of using this code: the user is allowed to spend a moderate amount of time in developing his/her own inner heuristic for the specific (sometimes tricky) application by only taking care of the "feasibility" viewpoint, and then let `TSpack` take care of the "optimization" viewpoint which means improve an initial (trivial) solution. Nevertheless, tuned versions of `TSpack` using effective inner heuristics have proved to be very effective for 2BP and 3BP, often finding state of the art results [13,14]. These results are summarized in Table 1.

In particular, Table 1 reports in the 2BP part the results obtained with 60 CPU seconds on a Silicon Graphics INDY R10000sc (195 MHz). For each pair instance size (n) - instance type (Class[1]), the average, over ten instances, number of bins used by the inner heuristic *Alternate Directions* (AD) [13], and by the Tabu Search are reported. The 3BP part of Table 1 is constructed exactly as the previous one and reports results obtained with 60 CPU seconds on a Digital Alpha 533 MHz, and using as inner heuristic algorithm *Height first-Area second* (HA) [14].

4. Extensions

As already mentioned, several basic extensions of 2BP and 3BP, as those allowing item rotation or requiring guillotine packing, may be directly solved by TSpack by simply providing a suitable inner heuristic. In this section we discuss further extensions of the code for the solution of other multi-dimensional packing problems.

The first family of problems we consider involves single-bin packing. Two main problems arise, known as the *Strip Packing Problem* in the two-dimensional case and the *Container Loading Problem* in the three-dimensional one, in which there is a unique bin, whose size is infinite in one dimension (say, the height)

[1] Classes 1-4 are the ones proposed by Martello and Vigo [18], while classes 5-10 were proposed by Berkey and Wang [2], see e.g., [13] for details.

Table 1 Tabu Search results for Multi-Dimensional Bin Packing Problems [13,14].

2BP: results from [13], $k = 3$.

n	Class	AD	TS(AD)	n	Class	AD	TS(AD)	n	Class	AD	TS(AD)	n	Class	AD	TS(AD)	n	Class	AD	TS(AD)
20	1	5.8	5.5	40	1	11.8	11.4	60	1	16.6	16.2	80	1	23.7	23.2	100	1	27.9	27.7
	2	6.2	5.8		2	12.0	11.4		2	16.9	16.2		2	23.6	22.6		2	28.9	28.4
	3	14.4	14.3		3	27.9	27.8		3	44.0	43.8		3	57.8	57.7		3	69.9	69.5
	4	4.3	4.3		4	7.7	7.5		4	10.7	10.4		4	13.5	13.0		4	16.9	16.6
	5	7.5	7.1		5	13.9	13.5		5	20.7	20.1		5	28.6	28.2		5	33.1	32.6
	6	1.0	1.0		6	2.0	2.0		6	2.7	2.7		6	3.3	3.3		6	4.0	4.0
	7	5.5	5.5		7	10.1	9.7		7	15.0	14.0		7	20.3	19.8		7	23.6	23.6
	8	1.0	1.0		8	1.9	1.9		8	2.6	2.6		8	3.3	3.3		8	3.8	3.8
	9	6.8	6.6		9	12.5	11.9		9	18.9	18.2		9	25.8	25.1		9	29.8	29.5
	10	1.0	1.0		10	1.9	1.9		10	2.2	2.2		10	3.0	3.0		10	3.4	3.4

3BP: results from [14], $k = 4$.

n	Class	HA	TS(HA)	n	Class	HA	TS(HA)	n	Class	HA	TS(HA)	n	Class	HA	TS(HA)
50	1	13.9	13.4	100	1	27.6	27.1	150	1	38.1	37.3	200	1	52.7	51.9
	2	14.2	13.8		2	27.0	25.8		2	38.7	37.5		2	51.4	50.5
	3	14.0	13.4		3	27.1	26.3		3	39.1	38.1		3	52.1	50.6
	4	29.4	29.4		4	59.0	59.0		4	86.9	86.8		4	119.0	118.8
	5	8.5	8.4		5	15.8	15.1		5	21.4	20.7		5	28.6	28.0
	6	10.5	9.9		6	20.0	19.3		6	30.6	29.7		6	39.1	38.2
	7	8.0	7.5		7	13.3	12.6		7	17.2	16.5		7	25.2	24.6
	8	9.9	9.3		8	19.9	19.0		8	25.7	24.6		8	31.6	31.0

and finite in the remaining one(s). The objective is to allocate all the items by minimizing the height to which the bin is used.

In the second family of problems, known as *Multi-Dimensional Knapsack Problems*, there is a single multi-dimensional bin and each item is associated with a given numerical value (*profit*). The objective is to pack a subset of items whose total profit is a maximum.

The above single-container problems cannot be directly solved by TSpack. However, the code can be modified so as to handle a relevant variant of both families, namely the one requiring that the items are packed in rows forming layers (see Figure 4). This kind of packing is known as *level packing* in the bin/strip context, and as *2-* and *3-staged cutting* in the knapsack context (see Lodi, Martello and Vigo [16]). A relevant aspect of these variants is that the resulting packings can be automatically separated through guillotine cuts, as frequently required in several industrial applications.

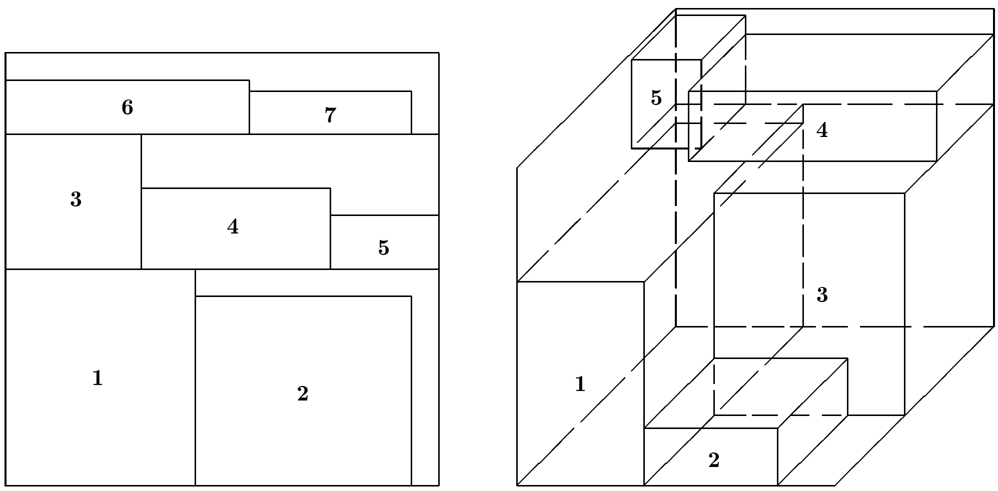

Figure 4. Two- and Three-Dimensional level packings.

Since the levels can be seen as separate bins of different heights, single-bin level-packing problems can be solved by modifying TSpack so as to pack the items in separate levels rather than into bins. For the first family of problems, the current set of levels may then be aggregated by simply placing them one above the other. For the second family instead, it is necessary to select a subset of the obtained levels. This can be done by solving (either exactly or heuristically) an associated one-dimensional knapsack problem in which: (i) there is an "element" per level, whose profit is the sum of the profits of the items it packs, while its weight is the level height; (ii) the knapsack capacity is the height of the bin.

Acknowledgements

The authors warmly thank John Sozinov for downloading their code from the web, testing it, and helping in fixing bugs.

References

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

[2] J.O. Berkey and P.Y. Wang. Two dimensional finite bin packing algorithms. *Journal of the Operational Research Society*, 38:423–429, 1987.

[3] C.S. Chen, S.M. Lee, and Q.S. Shen. An analytical model for the container loading problem. *European Journal of Operational Research*, 80:68–76, 1995.

[4] F.K.R. Chung, M.R. Garey, and D.S. Johnson. On packing two-dimensional bins. *SIAM Journal of Algebraic and Discrete Methods*, 3:66–76, 1982.

[5] K.A. Dowsland and W.B. Dowsland. Packing problems. *European Journal of Operational Research*, 56(1):2–14, 1992.

[6] H. Dyckhoff and U. Finke. *Cutting and Packing in Production and Distribution*. Physica Verlag, Heidelberg, 1992.

[7] H. Dyckhoff, G. Scheithauer, and J. Terno. Cutting and Packing (C&P). In M. Dell'Amico, F. Maffioli, and S. Martello, editors, *Annotated Bibliographies in Combinatorial Optimization*. John Wiley & Sons, Chichester, 1997.

[8] J.B. Frenk and G.G. Galambos. Hybrid next-fit algorithm for the two-dimensional rectangle bin-packing problem. *Computing*, 39:201–217, 1987.

[9] D. S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, MIT, Cambridge, MA, 1973.

[10] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing probles: A survey. *European Journal of Operational Research*, 141:3–13, 2002.

[11] A. Lodi, S. Martello, and D. Vigo. Neighborhood search algorithm for the guillotine non-oriented two-dimensional bin packing problem. In S. Voss, S. Martello, I.H. Osman, and C. Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 125–139. Kluwer Academic Publishers, Boston, 1998.

[12] A. Lodi, S. Martello, and D. Vigo. Approximation algorithms for the oriented two-dimensional bin packing problem. *European Journal of Operational Research*, 112:158–166, 1999.

[13] A. Lodi, S. Martello, and D. Vigo. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS Journal on Computing*, 11:345–357, 1999.

[14] A. Lodi, S. Martello, and D. Vigo. Heuristic algorithms for the three-dimensional bin packing problem. *European Journal of Operational Research*, 141:171–181, 2002.

[15] A. Lodi, S. Martello, and D. Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123:379–396, 2002.

[16] A. Lodi, S. Martello, and D. Vigo. Models and bounds for two-dimensional level packing problems. *Journal of Combinatorial Optimization*, 2002 (to appear).

[17] S. Martello, D. Pisinger, and D. Vigo. The three-dimensional bin packing problem. *Operations Research*, 48:256–267, 2000.

[18] S. Martello and D. Vigo. Exact solution of the two-dimensional finite bin packing problem. *Management Science*, 44:388–399, 1998.

[19] D. Pisinger, E. den Boef, J. Korst, S. Martello, and D. Vigo. Robot-packable and general variants of the three-dimensional bin packing problem. Technical report 01/05, DIKU, University of Copenhagen, 2001.

[20] G. Scheithauer. A three-dimensional bin packing algorithm. *J. Inform. Process. Cybernet.*, 27:263–271, 1991.