The 7th International Symposium on Operations Research and Its Applications (ISORA'08)
Lijiang, China, October 31–Novemver 3, 2008
Copyright © 2008 ORSC & APORC, pp. 292–302

A Tabu Search Based Algorithm for Cargo Loading Problem

Li Pan[1] Joshua Z. Huang[2] Sydney C.K. Chu[1]

[1]Department of Mathematics, University of Hong Kong, Hong Kong, China
[2]E-Business Technology Institute, University of Hong Kong, Hong Kong, China

Abstract Given a finite set of three-dimensional boxes in different sizes and an unlimited set of containers in the same size, the cargo loading problem is to determine the minimum number of containers that can contain all the boxes. The problem is NP-hard. In this paper, we propose to use Tabu search optimization with a tree-based heuristic cargo loading algorithm as its inner heuristic to solve this problem. This approach is more flexible in taking different box conditions into consideration. Experimental results have shown that the new approach could find better solutions on average than those by other recent meta- or heuristic algorithms.

Keywords Tabu search; cargo loading; bin packing; tree-based heuristic algorithm.

1 Introduction

Cargo loading or container loading is an important operation in modern logistics. Hundreds of containers are loaded daily at many large distribution centers and manufacture warehouses. Unfortunately, loading operations are still manual in many warehouses. Improvement on container capacity utilization and loading efficiency can significantly reduce logistics costs.

In warehouse logistics, cargo loading operation involves two problems: determining the minimum number of containers that are required and loading each container to better utilize its capacity. Mathematically, the cargo loading problem can be seen as a specific aspect of the three-dimensional bin packing problem (3BPP), i.e., allocating without overlapping a finite set of rectangular items (cargos) into bins (containers) so as to meet certain objectives. Some examples are:

1. Minimize the number of the bins to pack all items.
2. Maximize the total volume of items that can be packed into one bin.
3. Find a method to pack all given items into one bin.

In real applications, certain constraints are often added to the optimization process. For example, item orientation, spatial relationship, packing sequence, etc. These additional constraints further complicate the process to optimize the objectives. Generally speaking, the cargo loading problem is NP-hard. Therefore, heuristic methods are often used for its solution.

In this paper, we present a tree-based heuristic algorithm to load containers. We first vertically divide a container into layers and pack items into these layers. We treat each layer as a rectangle and transform the 3D packing problem into a 2D packing problem. Based on the B-tree structure [1], we recursively partition a layer into a set of smaller rectangles according to the size distribution of items to be packed and represent each rectangle as a node of a binary tree. Each item is allocated to a node and the packing location in the layer is recorded. As such, a tree with allocated items is a packing plan for the physical packing of the items. The major advantages of this algorithm are its efficiency and flexibility to consider different packing conditions. Therefore, it is suitable for real applications. After the trees for all layers are generated, we use a dynamic programming method to pack containers vertically and hence determine the initial number of containers that are needed to pack the given set of items.

We use Tabu search framwork [9] to further optimize the packing plan by repacking some containers with different subsets of items to reduce the number of containers that are finally needed. In this optimization process, utilization of capacity of some containers is increased. Experimental results have shown that the new approach could find solutions on average better than those by some other cargo loading methods.

The remainder of this paper is structured as follows: Section 2 presents the packing problem and related work. Section 3 introduces the tree-based heuristic algorithm for container loading. Section 4 describes the Tabu search process for optimization. Experiments and comparison analysis are given in Section 5. Finally, the paper is briefly concluded in Section 6.

2 Problem statement and related work

The cargo loading problem is a special case of the 3D bin packing problem (3BPP). The 3BPP problem can be simply stated as follows: Given a set of n three-dimensional rectangular items characterized by length l_i, width w_i and height h_i $(i = 1, 2, \ldots, n)$, and an unlimited set of identical three-dimensional bins of length L, width W and height H, find the minimal number of bins that can contain all items. The sides of items are packed in parallel with sides of the bins and items can be rotated 90° on the flat floor.

It is known that this problem is NP-hard, since even the one-dimensional bin packing problem is NP-hard [6]. Hence, heuristic methods are often used as alternatives to give "acceptable"solutions within a comparatively short time. Over the last two decades, there has been considerable advances in methods for solving a wide variety of 3BPP. In [12], the first exact branch-and-bound algorithm was proposed for 3BPP. Extensive computational results show that only small scale problems (less than 90 items) can be solved through this method within a reasonable time. When the problem size increases, the computational time of the exact method grows exponentially. Therefore, recent research on 3BPP is more focused on heuristic algorithms. First fit decreasing (FFD) and best fit decreasing (BFD) algorithms are two fast heuristic algorithms [4] that are often used as inner heuristics in new metaheuristic algorithms. In [9], a new approximation algorithm for packing bins with fixed item orientation was presented. It was used as a subordinate heuristic part within the Tabu Search metaheuristics. C code for this algorithm to solve two- or three-dimensional bin packing problems is given in [10]. Genetic algorithms, simulated annealing and some other heuristics are also used in solving the bin-packing problems

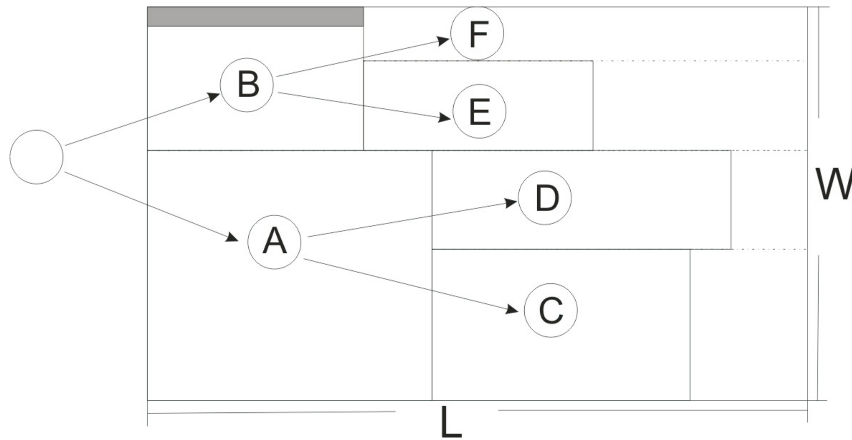

Figure 1: A layer packing plan tree.

[3, 5, 11].

3 B-tree algorithm

In this section we propose a binary tree heuristic algorithm that is used as the inner heuristic algorithm for the tabu search. The B-tree heuristic algorithm packs the items by layers which are vertical partitions of bins. The first layer lay on the base of the bin. A layer is viewed as a rectangle with a size of $L \times W$, while L and W are the length and the width of the rectangle which are equal to the inner length and width of the bin. To pack items in a layer, each item is also viewed as a rectangle of size $l_i \times w_i$, where l_i and w_i are the length and the width of the rectangle. Items are packed in parallel with the sides of the layer.

Given a set of items to be packed in a layer, the height of the layer is defined by the tallest item packed on this layer. The packing location of an item is defined by its lower left corner, denoted as p.

In the following, we present the binary tree representation of a packed layer and the heuristic algorithm for packing a layer.

3.1 B-tree representation of a layer packing plan

Given a set of items and a layer, the plan of packing these items in the layer can be represented as a binary tree as shown in Fig.1. Except for the root, each node represents an area that is partitioned from the layer. Items are packed in the areas of the nodes. The area of a parent node contains all areas of its child nodes. The difference between the parent node area and the child node areas is the area that an item is packed in the parent node.

Fig.2 shows the information being recorded in the root and nodes. The root node contains the height of the layer which is the height of the tallest item being packed. A node contains information about its parent, its children (two branches), the area (Packing Area) the node represents, the item packed, the packing position, and the remaining packing

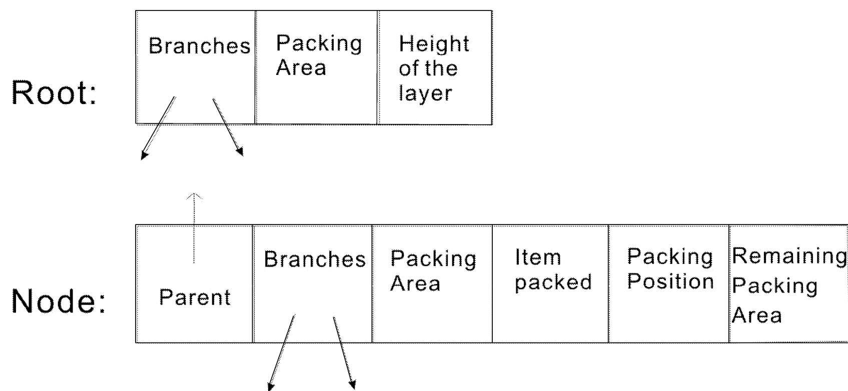

Figure 2: Information recorded on each node.

area, i.e., the area remained after packing the item. This remaining area is used as possible packing areas for its child nodes. If the node has not been packed, the remaining area is equal to the packing area, for instance node F in Fig.1.

In generating a B-tree from a set of items, say item i, there are four operations that can be performed at node j. As shown in Fig.3, if node j is packed and, after packing item i, the remaining area is still big enough to pack a new item, two child nodes are created, one for packing item i and one containing an area that is partitioned from the remaining area (Fig.3(a)). If node j is packed and the remaining area is not big enough to pack a new item after packing i, two child nodes are created, one containing the packed item with all the remaining area of node j as its packing area and one being a dummy node (Fig.3(b)). If node j is not packed and the remaining area is still big enough to pack a new item after packing item i, then pack i and create two child nodes with packing areas partitioned by the packed item from the remaining area of node j (Fig.3(c)). If node j is not packed and after packing i, the remaining area is not big enough to pack a new item, then leave this node as a leaf node (Fig.3(d)).

3.2 Packing method

For some classical finite Bin-packing algorithms, like Finite next-fit, Finite first-fit, etc. [2], the poor absolute worst case performance is mainly due to their level-oriented characteristics. Since the items are packed with their bottom edges on a level and the height of the level is defined by the tallest item packed on this level, some available areas are often wasted. The B-tree algorithm can reduce the wasted areas by adopting sub-branches instead of one level.

We pack the items with similar heights on the same layer so we first sort all the items on height and arrange the items in the order of decreasing heights. Then, we partition the items into clusters with different ranges of item heights. The clustering procedure is as follows. The first cluster contains the items whose height is taller than $\beta h_1 (\beta < 1)$ where h_1 is the height of the tallest item. Assume item i is the tallest item in the remaining items. The next cluster contains the items whose heights are taller than βh_i. The process continues until all items are clustered.

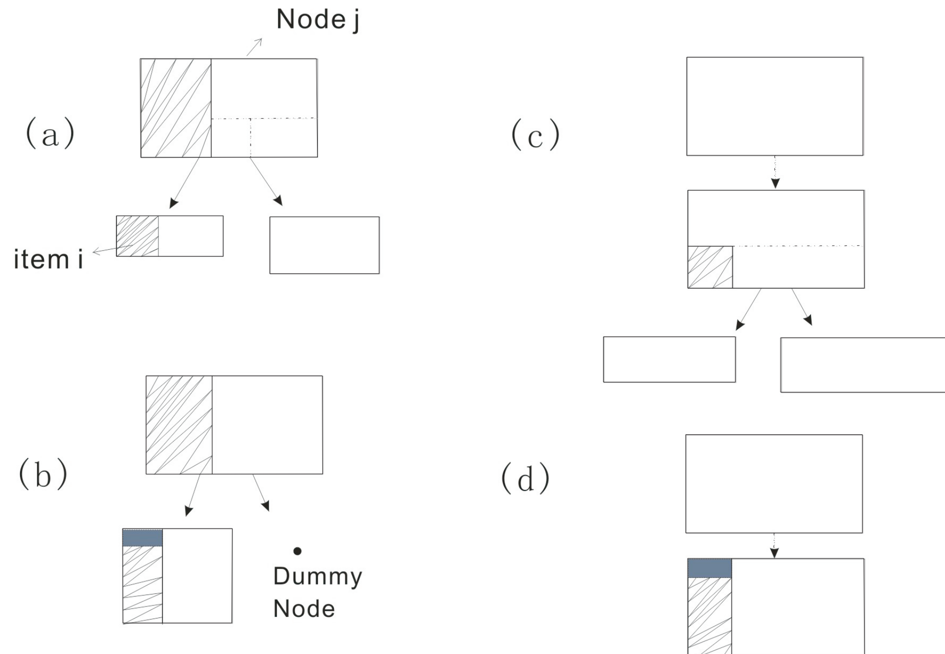

Figure 3: Packing and branching conditions without considering rotation: (a) One item has been packed on node j, and the remaining area can be further branched into two slices after packing i; (b) One item has been packed on node j, and the remaining area cannot be further branched; (c) No item has been packed on node j, and the remaining area can be further branched into two slices after packing i; (d) No item has been packed on node j, and the remaining area cannot be further branched.

In each cluster, the items are sorted by width and arranged from the widest item to the narrowest item. Starting from the widest item, the B-tree packing algorithm packs the items in the cluster one at a time. The items are packed in the increasing order of clusters.

To begin, we initialize a new tree with a root node and try to pack the first item at the lower left corner of the new layer. The first item divides the layer into two areas and two child nodes are created to represent the two packing areas as the left and the right child node of the root respectively. The first item is packed to the left child node. The lower left corner of the layer is recorded as the packing position of the first item. The remaining packing area is calculated as described below. The right child node only records the packing area that is not packed (like the situations in Fig.3(c) and 3(d)).

To pack a new item in the remaining areas of the nodes, the tree is traversed by the item across all the nodes that can pack the item and a node is identified that can pack the item with the minimal waste area. The depth-first traversal order is used in this process.

The waste area is calculated as follows: For item i with size of $(l_i \times w_i)$ to be pack to node j, the remaining area of node j is S_j ($S_j =< L1_j, L2_j > \times < W1_j, W2_j >$), where $L1_j, L2_j, W1_j, W2_j$ are the coordinates of the length side and the width side of S_j, $< L1_j, L2_j >$ represents the length segment and $< W1_j, W2_j >$ the width segment. To pack

item i to node j, the waste area WA for node j is calculated under the following conditions:

1. In case that item i is not rotated:
   (a) If an unpacked item k with size of $(l_k \times w_k)$ exists that can be packed in $< L1_j, L2_j > \times < W1_j + w_i, W2_j >$, then $WA = (L2_j - L1_j) \times (W2_j - W1_j - w_i - w_k)$;
   (b) If such an item does not exist, then $WA = l_i \times (W2_j - W1_j - w_i)$.
2. In case that item i is rotated:
   (a) If an unpacked item k with size of $(l_k \times w_k)$ exists that can be packed in $< L1_j, L2_j > \times < W1_j + l_i, W2_j >$, then $WA = (L2_j - L1_j) \times (W2_j - W1_j - l_i - w_k)$;
   (b) If such an item does not exist, $WA = w_i \times (W2_j - W1_j - l_i)$.

After all waste areas are calculated for all available nodes that can pack the item, the node with the minimal waste area is selected. The item is packed following the conditions below:

1. In case that one item has been packed on the selected node j (situations (a) and (b) in Fig.3):
   (a) If *minWA* is calculated from condition 1.(a), create two new child nodes from node j. Pack item i to the left node. Record the packing area as $< L1_j, L2_j > \times < W1_j, W1_j + w_i >$ and the remaining area as $< L1_j + l_i, L2_j > \times < W1_j, W1_j + w_i >$. For the right node, record only the packing and the remaining area as $< L1_j, L2_j > \times < W1_j + w_i, W2_j >$. Leave the item information empty.
   (b) If *minWA* is calculated from condition 1.(b), create two new child nodes from node j. Pack item i to the left node. Record the packing area as $< L1_j, L2_j > \times < W1_j, W2_j >$ and the remaining area as $< L1_j + l_i, L2_j > \times < W1_j, W2_j >$. For the right node, record the remaining area as $< 0, 0 > \times < 0, 0 >$ to create a dummy node.
   (c) If *minWA* is calculated from condition 2.(a), create two new child nodes from node j. Pack item i to the left node. Record the packing information of node i, the packing area as $< L1_j, L2_j > \times < W1_j, W1_j + l_i >$ and the remaining area as $< L1_j + w_i, L2_j > \times < W1_j, W1_j + l_i >$. For the right node, record only the packing and the remaining area as $< L1_j, L2_j > \times < W1_j + l_i, W2_j >$. Leave the item information empty.
   (d) If *minWA* is calculated from condition 2.(b), create two new child nodes from node j. For the left node, record the packing information, the packing area as $< L1_j, L2_j > \times < W1_j, W2_j >$ and the remaining area as $< L1_j + w_i, L2_j > \times < W1_j, W2_j >$. For the right node, record the remaining area as $< 0, 0 > \times < 0, 0 >$ to create a dummy node.
2. In case that no item has been packed on the selected node j (situations (c) and (d) in Fig.3):
   (a) If *minWA* is calculated from condition 1.(a), add packing information of item i to node j, and create two new child nodes from node j. For the left node,

record the packing area and the remaining area as $< L1_j + l_i, L2_j > \times < W1_j, W1_j + w_j >$. For the right node, record the packing area and the remaining area as $< L1_j, L2_j > \times < W1_j + w_i, W2_j >$. Both child nodes record no packing information.

(b) If *minWA* is calculated from condition 1.(b), add packing information of item i to node j and do not create child nodes.

(c) If *minWA* is calculated from condition 2.(a), add packing information of item i to node j, and create two new child nodes from node j. For the left node, record the packing area and the remaining area as $< L1_j + w_i, L2_j > \times < W1_j, W1_j + l_i >$. For the right node, record the packing area and the remaining area as $< L1_j, L2_j > \times < W1_j + l_i, W2_j >$. Both child nodes record no packing information.

(d) If *minWA* is calculated from condition 2.(b), add packing information of item i to node j and do not create child nodes.

Item i will be packed at position p in order of preference:

1. In an initialized layer of height bigger than or equal to h_i, at position p with the smallest WA;
2. In an initialized layer of height less than h_i, at position p with the smallest WA and updating the layer's height;
3. Create a new layer (tree), with its lower left corner packed in the lower left corner of the new layer.

3.3 Bin packing

After packing all items into layers, we need to pack all layers into bins. Let $H_1, H_2, \ldots, H_s$ represent s different heights in n layers. n_j layers have height H_j. A finite bin packing solution is obtained by a dynamic programming technique as follows.

Assume we have n layers with s different heights $H_1, H_2, \ldots, H_s$ to be packed into the containers. Let H be the height of containers and H_i the height of n_i layers with the same height. The bin packing problem can be formulated to a one-dimensional multiple length cutting problem with only one length. A cutting pattern is defined if we can find integers $a_i \geq 0, i = 1, \ldots, s$ such that $a_1 H_1 + a_2 H_2 + \cdots + a_s H_s \leq H$. Suppose that the jth cutting pattern generates a_{ij} pieces of length H_i. Let x_j be the number of times the jth cutting pattern is used. The problem is formulated as follows:

$$\begin{aligned} \text{Minimize} \quad & c = x_1 + x_2 + \cdots + x_k \\ \text{Subject to} \quad & a_{i1} x_1 + a_{i2} x_2 + \cdots + a_{ik} x_k \geq n_i, i = 1, \ldots, s \\ & x_j \geq 0,\ x_j \text{ integer } j = 1, \ldots, k \end{aligned}$$

where k is the total number of possible cutting patterns. The method to solve this problem is to cut the first length H using dynamic programming to minimize the wastage left. Then, we cut the second length H and so on until all required pieces are cut. During the Nth cutting procedure, let y_i be the number of the ith group of layers created in the cutting procedure, and R_i the number of pieces of layer i remained to be cut after $N - 1$ cuttings. The sub-problem for each cutting procedure can be formulated as follows:

$$\begin{array}{ll} \text{Maxmize} & y_1H_1 + y_2H_2 + \cdots + y_sH_s \\ \text{Subject to} & y_1H_1 + y_2H_2 + \cdots + y_sH_s \leq L \\ & y_i \leq R_i, i = 1, \ldots, s \\ & y_i \text{ non-negative integers} \end{array}$$

This can be solved by dynamic programming as:

$$f_l(x) = max\{\sum_{i=1}^{l} y_iH_i : \sum_{i=1}^{l} y_iH_i \leq x, y_i \in \{0,1,\ldots,R_i\}\}$$
$$P_l(x) = \text{number of type } l \text{ used in } f_l(x)$$

where

$$f_0(x) = 0, \text{ for all } x$$
$$f_l(0) = 0, \text{ for all } l$$
$$f_l(x) = \max_{i \in \{0,1,\ldots,min\{R_j,[x/H_l]\}\}} \{f_{l-1}(x - i * H_l) + i * H_l\}$$
$$P_l(x) = \text{value of } i \text{ used in } f_l(x)$$

Repeatedly solving this sub-problem until all $R_i = 0$ gives the final solution.

4 A tabu search approach

The metaheuristic method is a good way for guiding the operations of a subordinate heuristic to find a better solution for complicated combinatorial problems. In this paper, we use a tabu search metaheuristic technique. Reader is referred to [7] for an introduction to the tabu search algorithm. The approach is based on [9], in which the search scheme and neighborhood size are independent of the specific problem to be solved and the neighborhood size and structure are varied dynamically during the search procedure.

The initial upper bound solution is obtained by executing the B-tree heuristic algorithm. For a given set of items S, let $B(S)$ represent the output solution of executing B-tree heuristic algorithm on S. The search is started by packing each item into a separate bin. During the next steps, the target of search is trying to reduce the number of bins used. Here, let n^c represent the number of bins used in the current solution.

The target is reached by repacking a set of items S, trying to empty a target bin to improve the current solution. The target bin is chosen on the fact that, if a bin contains less and/or smaller items, it is more likely to be emptied through local optimization. We adopted the same policy used in [9], i.e., the target bin S_t is defined as the one which minimizes the filling function (α is a prefixed non-negative parameter).

$$\varphi(S_i) = \alpha \frac{\sum_{j \subseteq S_i} l_j w_j h_j}{LWH} - \frac{|S_i|}{n} \quad (1)$$

in which, n is the size of the problem. The first part of the formula represents the space utilization when $\alpha = 1$ and the second part is the ratio of the number of items packed in bin i to the number of all items.

At each iteration, we construct one set S, which is combined by one item j from target bin S_t and other k bins in the current solution. The new packing for set S is obtained by executing the B-tree algorithm B on S. The parameter k represents the size of the searching neighborhood, which is dynamically changed during the search procedure.

If $B(S) \leq k$, we can remove item j out of target bin S_t without creating more bins. Then, we update the packing solution n^c, reduce k by one unit, select another item from S_t to construct a new set S, and apply B-tree algorithm on them. Otherwise, S is reconstructed by a different set of k bins in the current solution; or when all the possible combinations of k-tuple bins have been considered for item j, we select another item in the target bin and apply B-tree algorithm on them. There are a tabu list and a tabu tenure $\tau_k(k = 1, \ldots, k_{max})$ for each neighborhood size k. Each tabu list stores the filling function values for the last τ_k moves performed in the corresponding k.

The search procedure is stopped if all item j from the target bin and all combination conditions of k-tuple bins in the current solution have been considered without obtaining any acceptable moves, or m moves have been run without improving packing solution (m is a prefixed positive integer parameter). For both cases, if k is less than a prefixed limit, increase k by one unit and explore the new neighborhood; otherwise, apply a diversification procedure.

There are two kinds of diversification actions: the first selecting the target bin with the second smallest value of the filling function; the second repacking items from the $\lfloor n^c/2 \rfloor$ bins with the smallest filling function value into separate bins. For both cases, empty all the tabu list and reset $k = 1$.

5 Computational experiments

In this section, we present the results of this new algorithm from computational experiments. The results have shown that within a short computation time this algorithm produced better solutions on average than other meta- or heuristic algorithms we have compared.

The algorithm was coded in C and run in a Centrino Duo processors personal computer. The test data was generated in reference with the data generation methods published in [12]. For the B-tree algorithm and tabu search approach the parameters were set to the following values: $\beta = 0.5, \alpha = 3.5, k_{max} = 3, \tau = 20$ and $m = 20$.

5.1 Test data

Six classes of test data were generated with the same method as in [12]. The data sets contained 50 to 150 items. The distributions of the items in these data sets were:

- Type 1: the majority of items are very high and long.
- Type 4: the majority of items are big.
- Type 5: the majority of items are small.
- Type 6: items with dimensions randomly generated in a small interval.
- Type 7: items with dimensions randomly generated in a medium interval.
- Type 8: items with dimensions randomly generated in a large interval.

We did not generate data of Types 2 and 3 as in [12], because the items can be rotated on the flat surface, meaning that these two types are similar to Type 1. For each type and each number of items (i.e., 50, 100, 150), ten data sets were generated.

To evaluate the tabu search based heuristic algorithm (TSBH), we compared the results with the B-tree (only) algorithm presented in Section 3, the guided local search heuristic algorithm (GLS) [8], the HA and HATS algorithms [9] and the MPV algorithm

[12]. GLS iteratively decreases the number of bins used by searching for feasible packing of bins. HATS is another tabu search algorithm using HA, a two-phase heuristic algorithm, as its inner heuristic algorithm. The MPV algorithm reaches an exact solution if it gets sufficient time. In the next section, we present the computational results.

5.2 Test results

The results of averages of bins used are summarized in Table 1. The first three columns give the problem class (with bin size) and the number of items n. For each algorithm, we report the average number of bins used from ten data set. TSBH was run with a time limit of 120 CPU seconds. GLS and MPV were run with a time limit of 1000 CPU seconds while HATS was run with a time limit of 60 CPU seconds. The computing times of B-tree algorithm and HA were negligible.

From Table 1, we can see that our results are satisfactory. The TSBH algorithm generally improved the initial deterministic solution produced by the inner heuristic B-tree. For most cases, TSBH found solutions similar to or better than other meta- or heuristic algorithms. Even though for some cases, the inner heuristic algorithm B-tree alone did not perform as well as HA, after combining with tabu search approach, the results of TSBH were generally better than those of HATS. Among TSBH, GLS and MPV, TSBH (and GLS) obtained equal or better solutions than the MPV algorithm for all except one class data, and a slightly better average solution than GLS, with far less time. On average, the TSBH algorithm used 18.27 bins which was the minimum among all the above algorithms and was very comparable to GLS.

Class	Bin Size	n	TSBH	B-tree	GLS[8]	HATS[9]	HA[9]	MPV[12]
1	100	50	13.0	15.6	13.4	13.4	13.9	13.6
	×	100	23.8	27.9	26.7	27.1	27.6	27.3
	100	150	37.9	42.8	37.0	37.3	38.1	38.2
4	100	50	29.1	29.1	29.4	29.4	29.4	29.4
	×	100	58.3	58.4	59.0	59.0	59.0	59.1
	100	150	85.1	85.1	86.8	86.8	86.9	87.2
5	100	50	7.0	7.7	8.3	8.4	8.5	9.2
	×	100	15.1	17.0	15.1	15.1	15.8	17.5
	100	150	21.0	23.4	20.2	20.7	21.4	24.0
6	10	50	9.8	10.6	9.8	9.9	10.5	9.8
	×	100	19.1	20.3	19.1	19.3	20.0	19.4
	10	150	29.1	30.6	29.4	29.7	30.6	29.6
7	40	50	6.8	7.6	7.4	7.5	8.0	8.2
	×	100	13.9	16.0	12.3	12.6	13.3	15.3
	40	150	17.9	21.0	15.8	16.5	17.2	19.7
8	100	50	8.9	10.6	9.2	9.3	9.9	10.1
	×	100	18.0	20.8	18.9	19.0	19.9	20.2
	100	150	24.4	28.4	23.9	24.6	25.7	27.3
Total			438.5	473.2	441.7	445.6	455.7	465.1
Average			18.27	19.72	18.40	18.57	18.99	19.38

Table 1: Experimental results: averages of ten instances for each class and size.

6 Conclusion

The cargo loading problem is a specific problem of the three-dimensional bin packing problems. It's NP-hard and computationally difficult to solve. We have presented a Tabu search based method with a B-tree heuristic algorithm as its inner heuristics. Computational experiments have shown that the new method is effective and on average yields good results compared with some recent meta- or heuristic algorithms.

References

[1] Aho, A.V., Hopcroft, J.E., Ullman, J.D., *Data structures and algorithms.* Reading, MA: Addison-Wesley. (1983)

[2] Berkey, J.o., Wang, P.Y.,*Two dimensional finite bin packing algorithms.* Journal of Operational Research Society, **38**, 423-329 (1987)

[3] Bortfeldt, A., Mack, D., *A heuristic for the three-dimensional strip packing problem.* European Journal of Operational Research, **183**, 1267-1281 (2005)

[4] Dyckhoff, H., Toth, P., *Knapsack Problems: Algorithms and Computer Implementations.* Chichester: Wiley (1990)

[5] Gancavels, J.F., *A hybrid genetic algorithm-heuristic for a two-dimensional orthogonal packing problem.* European Journal of Operational Research. **183**, 1212-1238 (2002)

[6] Gary, M., Johnson, D., *Computers and intractability: a guide to the theory of NP-completeness.* San Francisco: W.H. Freeman (1979)

[7] Glover, F., Laguna, M., *Tabu search.* Boston, MA: Kluwer Academic Publisher. (1997)

[8] Faroe, O., Pisinger, D., Zachariasen, M., *Guided local search for the three-dimensional bin-packing problem.* INFORMS Journal on Computing. **15(3)**, 267-283 (2003)

[9] Lodi, A., Martello, S., Vigo, D., *Heuristic algorithms for the three-dimensional bin packing problem.* European Journal of Operational Research. **141**, 410-420 (2002)

[10] Lodi, A., Martello, S., Vigo, D., *Tspack: A unified tabu search code for multi-dimensional bin packing problems.* Annals of Operations Research. **131**, 203-213 (2004)

[11] Loh, K., Golden, B., Wasil, E., *Solving the one-dimensional bin packing problem with a weight annealing heuristic.* Computers & Operations Research. **35**, 2283-2297 (2006)

[12] Martello, S., Pisinger, D., Vigo, D., *The three-dimensional bin packing problem.* Operations Research, **48**, 256-267 (2000)