Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Operations Research 32 (2005) 2051–2058
computers & operations research
www.elsevier.com/locate/dsw

A two-dimensional vector packing model for the efficient use of coil cassettes

Soo Y. Chang[a,*], Hark-Chin Hwang[b], Sanghyuck Park[a]

[a] *Department of Industrial Engineering, Division of Electrical and Computer Engineering, Pohang University of Science and Technology, Hyoja Dong San 31, Pohang, Kyungbuk 790-784, Republic of Korea*
[b] *Department of Industrial Engineering, Chosun University, 375 Seosuk-Dong, Dong-Gu, Gwangju 501-759, Republic of Korea*

Abstract

We consider the problem of efficiently packing steel products, known as *coils*, into special containers, called *cassettes* for shipping. The objective is to minimize the number of cassettes used for packing all the given coils where each cassette has capacity limits on both total payload weight and size. We model this problem as a *two-dimensional vector packing problem* and propose a heuristic. We also analyze the worst-case performance of the proposed algorithm under a special condition which, in fact, holds for the particular real-world case that we handled. Our computational experiment with real production data shows that the proposed algorithm performs quite satisfactorily in practice.

Keywords: Two-dimensional vector packing; Worst-case analysis

1. Introduction

The key determinant of the seaport operation efficiency is the time required for loading and unloading. At the P Steel company, a dramatic time reduction is achieved through the use of *roll-on roll-off* system where the rolls of thin sheet of steel, known as coils, are packed into the specially designed cassettes and the loaded cassettes are rolled on and off the ship. Fig. 1 shows the cassette and coils loaded in it. In this paper, we consider the problem of formulating a plan for packing a given set of coils into the minimum possible number of cassettes.

As shown in Fig. 1, each cassette has identical width, length and total payload limit, whereas each coil has three attributes; weight, width and diameter, as illustrated in Fig. 2. However, since

* Corresponding author.
E-mail address: syc@postech.ac.kr (S.Y. Chang).

0305-0548/$ - see front matter © 2004 Published by Elsevier Ltd.
doi:10.1016/j.cor.2004.01.006

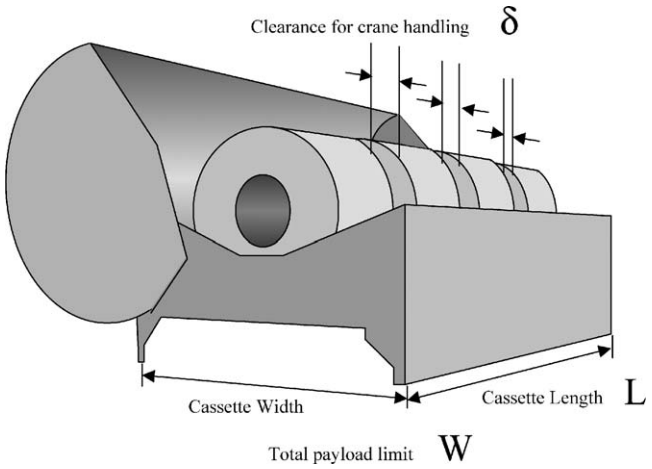

Fig. 1. Loaded cassette.

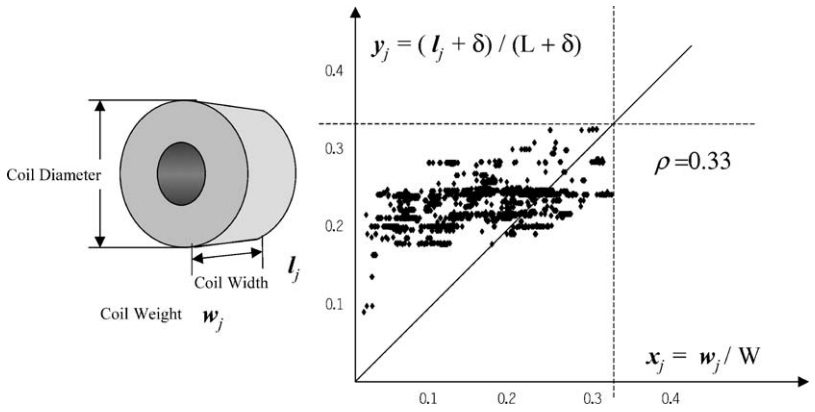

Fig. 2. Distribution of converted coil width and weight.

the width of cassette is designed wide enough to contain the coil with any size of diameter, we have to be concerned with only the weight and width of coils in packing them into each cassette.

Suppose that the payload weight limit and the length of the cassette is W and L, respectively. Also suppose that we are given n coils and the jth coil has its weight and width of w_j and l_j, respectively. Then, the sum of w_j of all the coils packed into the same cassette should not exceed W. Likewise, the total sum of l_j of the coils loaded in each cassette is also limited. Regarding this limitation, however, we must consider the minimum space required between every pair of coils loaded adjacent to each other, denoted by δ. In particular, as illustrated in Fig. 1, $k-1$ times δ must be taken off from the length of the cassette in addition to the sum of l_j of the coils when k coils are packed into one cassette. One simple way to comply with this restriction is to check if the sum of $l_j + \delta$, instead of l_j, of all coils packed into the same cassette exceeds $L + \delta$, instead of L. Hence, for further simplification, we define converted weight and width as

$$x_j = w_j/W \quad \text{and} \quad y_j = (l_j + \delta)/(L + \delta) \quad \text{for } j = 1, \ldots, n. \tag{1}$$

Treating these converted weight and width as the coordinates in two-dimensional space, the problem of packing coils into minimum number of cassettes can be modelled as a *two-dimensional vector packing problem*, or 2DVPP for short, as described below. This problem is, in fact, a special case of the *d-dimensional vector packing problem* (d-DVPP) which was introduced by Garey et al. [1].

2DVPP

Given the set $J = \{1, \ldots, n\}$ *of items and* (x_j, y_j) *for each item* j *in* J, *partition* J *into minimum number of subsets such that the sum of* x_j *as well as the sum of* y_j *of the items in each subset never exceed* 1.

Yao [2] proves that there exists no $O(n \log n)$-time algorithm for d-DVPP which is guaranteed to generate a solution requiring no more than d times the optimum number of bins. Fernandez et al. [3] proposed an asymptotic polynomial time approximation scheme that can be generalized to yield a solution requiring no more than $d + \varepsilon$ times the optimum number of bins for d-DVPP. Later, this result is improved by Chekuri and Khanna [4] to become an algorithm for d-DVPP which is guaranteed to yield a solution requiring no more than $1 + \varepsilon d + O(\ln \varepsilon^{-1})$ times the optimum number of bins. For 2DVPP, Spieksma [5] proposes a branch-and-bound algorithm for the problem. Woeginger [6] shows that, even when $d = 2$, d-DVPP cannot have an asymptotic polynomial time approximation scheme unless P = NP. Caprara et al. [7] present an interesting discussion on difficulties involved in setting up a tight lower bound on the optimum number of bins. Recently, Kellerer et al. [8] present an $O(n \log n)$ time algorithm which is guaranteed to yield a solution requiring no more than twice the optimum number of bins.

Our literature survey seems to suggest that it may be quite unlikely to develop a 2-DVPP algorithm with the guarantee to yield a solution requiring strictly less than twice the optimum number of bins, unless the problem has some special structure that can be exploited. Fortunately, we noticed that the particular problem instances that we had to solve for the steel company, which is one of the world largest steel company producing over 25 million tons of steel product annually, happens to have a very special feature. Namely, the weight, w_j, and width, l_j, of every coil to be loaded on the cassette does not exceed a fixed fraction of the payload limit, W, and the length of the cassette, L, respectively. Hence, both x_j and y_j, defined from w_j and l_j as in (1), are no more than a constant fraction denoted by ρ which is about 0.33 for all the actual problem instances as illustrated in Fig. 2. Very much encouraged by this simple observation, we develop an algorithm with $O(n^2)$ running time, which can be seen as a parametric extension of the approximation algorithm proposed in [8], and prove that it yields a solution requiring no more than $1/(1 - \rho)$, which is approximately 1.5 when ρ is about 0.33, times the optimum number of bins plus one. The tightness of this bound is still an open question and our computational experiment suggests that our algorithm would perform much better than its worst-case bound in practice.

2. Algorithm and analysis

We present the notation to be used in this paper, including the ones that are already defined in the previous section, for the sake of completeness. Recall that we let $J = \{1, 2, \ldots, n\}$ be the set of items and (x_j, y_j) be the size of the jth item, or item j. Without loss of generality, we assume

```
Algorithm Hedging
begin
  i := 1; S_i := ∅;
  J^x := X(J); J^y := Y(J);
  begin while((x(S_i) ≥ y(S_i) and J^y ≠ ∅) or (x(S_i) < y(S_i) and J^x ≠ ∅))
    if (x(S_i) ≥ y(S_i))
    then remove an arbitrary item j from J^y;
         if (x(S_i) + x_j > 1) find k ∈ S_i such that x(S_i) − y(S_i) ≤ x_k − y_k
                              and move the item k from S_i to J^x;
    else remove an arbitrary item j from J^x;
         if (y(S_i) + y_j > 1) find k ∈ S_i such that y(S_i) − x(S_i) ≤ y_k − x_k
                              and move the item k from S_i to J^y;
    add the item j to S_i;
    if (x(S_i) > 1 − ρ and y(S_i) > 1 − ρ)  i := i + 1;  S_i := ∅;
  end while
  if(J^x ≠ ∅) Pack By x_j;
  else Pack By y_j;
end

Pack By x_j( or y_j )
begin
  while (J^x ( or J^y ) ≠ ∅)
    pick an arbitrary item j from J^x ( or J^y );
      if ( x(S_i) + x_j( or y(S_i) + y_j)  > 1) i := i + 1; S_i := ∅;
      move the item j from J^x( or J^y ) to S_i;
  end while
end
```

Fig. 3. Algorithm hedging.

that the bin capacity in each dimension is 1, since we can always redefine the size of each item as in (1).

$\langle S_1, S_2, \ldots, S_p \rangle$ is used to represent a solution with items packed into p separate bins, where each S_i denotes the set of items in the ith bin. For any given set of items, say S, x-weight and y-weight of S, denoted by $x(S)$ and $y(S)$, are defined to be $x(S) = \Sigma_{j \in S} x_j$ and $y(S) = \Sigma_{j \in S} y_j$, respectively. Also, $X(S)$ and $Y(S)$ are defined as subsets of S defined as $X(S) = \{j \mid j \in S, x_j \geqslant y_j\}$ and $Y(S) = \{j \mid j \in S, x_j < y_j\}$, respectively.

Relative to ρ, we say that S_i is *x-complete* if $1-\rho < x(S_i) \leqslant 1$ and *y-complete* if $1-\rho < y(S_i) \leqslant 1$. If S_i is both x-complete as well as y-complete, we simply say that S_i is *complete*.

The proposed algorithm, that we call Hedging, is presented in Fig. 3. Note that the while-loop packs items into bins and creates a new bin when the current bin S_i becomes *complete*.

The while-loop is executed as far as either J^y is non-empty when $x(S_i) \geqslant y(S_i)$ or J^x is non-empty when $x(S_i) < y(S_i)$. Note that we can never exit the while-loop when both J^x and J^y are non-empty. Hence, there are three possible cases in which we exit the while-loop. The case when both J^x and J^y become empty would be the first case where no further work is done. The case when $J^y = \emptyset$ and $J^x \neq \emptyset$ but $x(S_i) \geqslant y(S_i)$ would be the second case when **Pack By** x_j is invoked to pack remaining items of J^x into the bins while creating new bins when S_i becomes x-complete. The third case is

when $J^x = \emptyset$ and $J^y \neq \emptyset$ but $x(S_i) < y(S_i)$ where **Pack By** y_j is invoked to pack remaining items of J^y into the bins while creating new bins when S_i becomes y-complete. Note that every bin except the last one ever packed by **Pack By** x_j and **Pack By** y_j must become x-complete and y-complete, respectively.

In order to establish the correctness of the algorithm, we need to make sure a few things stated in the following lemmas. First, we prove the following lemma.

Lemma 1. *When the set S_i constructed by the algorithm is non-empty, if $x(S_i) \geqslant y(S_i)$, we can always find an item k such that $x(S_i) - y(S_i) \leqslant x_k - y_k$.*

Proof. For notational ease, let S_i be $\{1, 2, \ldots, m\}$ and the item m be the last element added to S_i by the algorithm.

We prove the lemma by an inductive argument. So, we first note that the lemma trivially holds when $m = 1$. Then, supposing that the lemma holds for $1 \leqslant m \leqslant K$, for a positive integer K, we prove the lemma for $m = K + 1$.

When $m = K + 1$, define $S_i \setminus \{m\}$ as S_i' and note that S_i' is non-empty, since $|S_i'| = K > 0$. Then, we consider two cases, i.e., $m \in J^x$ and $m \in J^y$.

When $m \in J^x$, $x(S_i') < y(S_i')$ must hold due to the nature of the algorithm. Then, since $S_i' = S_i \setminus \{m\}$ $x(S_i) - x_m < y(S_i) - y_m$ must hold. Hence, the lemma follows, since m may be chosen as k in this case.

When $m \in J^y$, we have $x(S_i') \geqslant y(S_i')$, due to the nature of the algorithm. Then, our inductive presupposition ensures that we can find an item k in S_i' satisfying $x(S_i') - y(S_i') \leqslant x_k - y_k$. However, since $S_i' = S_i \setminus \{m\}$, we have $x(S_i) - y(S_i) = (x(S_i') + x_m) - (y(S_i') + y_m)$. Then, since $m \in J^y$ implies $x_m < y_m$, $x(S_i) - y(S_i) < x(S_i') - y(S_i')$ must hold. Hence, we have $x(S_i) - y(S_i) \leqslant x_k - y_k$ proving the lemma. □

Based on Lemma 1, we establish the following lemma.

Lemma 2. *When S_i, constructed by the algorithm, is non-empty but* **not complete** *and satisfies $x(S_i) \geqslant y(S_i)$, if the set J^y has an item j satisfying $x(S_i) + x_j > 1$, the set $S_i \setminus \{k\} \cup \{j\}$ must become* **complete** *if the item k is selected as defined in Lemma* 1.

Proof. Let S_i' be $S_i \setminus \{k\}$. Then, $x(S_i) + x_j > 1$ implies $x(S_i') + x_k + x_j > 1$. But since $x_k \leqslant \rho$, we have

$$x(S_i') + x_j > 1 - \rho. \tag{2}$$

However, from Lemma 1, we have $x(S_i) - y(S_i) \leqslant x_k - y_k$ which implies $y(S_i') \geqslant x(S_i')$. But, since $y_j > x_j$ for $j \in J^y$, we have

$$y(S_i') + y_j > x(S_i') + x_j. \tag{3}$$

Hence, (2) and (3) imply

$$y(S_i') + y_j > 1 - \rho. \tag{4}$$

On the other hand, $y(S_i) \leqslant 1-\rho$ must hold, since, $x(S_i) \geqslant y(S_i)$ and S_i is not complete. Hence, we have $y(S_i') \leqslant y(S_i) \leqslant 1-\rho$. Then, since $y_j \leqslant \rho$, we have

$$y(S_i') + y_j \leqslant 1. \tag{5}$$

Then by (3) and (5), we have

$$x(S_i') + x_j \leqslant 1. \tag{6}$$

Hence, the lemma follows from (2), (4), (5) and (6). □

Then, we have the following lemmas which we present without proof, since they can be proved by essentially the same arguments used for the proofs for Lemmas 1 and 2.

Lemma 3. *When the set S_i constructed by the algorithm is non-empty, if $y(S_i) > x(S_i)$, we can always find an item k such that $y(S_i) - x(S_i) \leqslant y_k - x_k$.*

Lemma 4. *When S_i, constructed by the algorithm, is non-empty but* **not complete** *and satisfies $y(S_i) > x(S_i)$, if the set J^x has an item j satisfying $y(S_i) + y_j > 1$, the set $S_i \setminus \{k\} \cup \{j\}$ must become* **complete** *if the item k is selected as defined in Lemma 3.*

Note that it takes as many as $|S_i| \leqslant n$ steps to locate the item k as defined in Lemmas 1 or 3. Also, note that one item is packed into S_i during the execution of each while-loop, except when we need to locate and remove the item k as defined in Lemmas 1 or 3, in which case, however, Lemmas 2 and 4 ensures that a new bin must be created. Therefore, total number of while-loop execution is bounded by the number of items n plus the maximum possible number of bins which is bounded by n. Therefore, it is clear that the algorithm Hedging takes no more than $O(n^2)$ steps.

In order to establish the worst-case performance bound of the proposed algorithm, we define C^{Hedging} and C^* to be the number of bins of the solution generated by our algorithm and the optimum solution, respectively. Then, based on Lemmas 2 and 4, we establish the worst-case performance bound as below.

Theorem 5. $C^{\text{Hedging}} \leqslant C^*/(1-\rho) + 1$.

Proof. Lemmas 2 and 4 assure that every S_i except the last one constructed in the while-loop is guaranteed to be complete, meaning both x-complete and y-complete.

After we exit from the while-loop, there are three possible cases.

Case 1($J^x = J^y = \emptyset$): In this case, no further computation is done. Hence, every S_i except the last one packed by the algorithm itself is complete.

Case 2($J^x \neq \emptyset$ and $x(S_i) \geqslant y(S_i)$): In this case, **Pack By** x_j is invoked starting with the current S_i satisfying $x(S_i) \geqslant y(S_i)$. Since every item j in J^x, $x_j \geqslant y_j$, every S_i except the last one packed by **Pack By** x_j must become x-complete at least. Hence, all S_i except the last one packed by the algorithm are x-complete, at least.

Case 3($J^y \neq \emptyset$ and $x(S_i) < y(S_i)$): Due to the similar argument as in CASE 2, all S_i except the last one packed by the algorithm are y-complete at least.

Therefore, each and every S_i except the last one in above three cases is complete, at least x-complete and at least y-complete in Case 1, 2 and 3, respectively. Hence, the theorem follows from

$$(C^{\text{Hedging}} - 1) * (1 - \rho) \leqslant C^*. \quad \square$$

3. Computational experiments

The proposed algorithm is implemented in Microsoft Visual C++ on a Pentium III platform and tested with ten real-world problem instances. The number of items in each instance was about 250. Table 1 presents the summary of the test results. The column under the label Max x_j, Max y_j and ρ, respectively, shows the maximum of x_j and y_j of each problem instance and ρ, taken as maximum of Max x_j and Max y_j. Then, the next column shows

$$L_C = \max\left\{\left\lceil\sum_{j\in J} x_j\right\rceil, \left\lceil\sum_{j\in J} y_j\right\rceil\right\} \tag{7}$$

which is the lower bound to the minimum number of bins as proposed by Spieksma in [5]. Theorem 5 shows that C^{Hedging}/C^* cannot be more than $1/(1-\rho)+1/C^*$ which is again no more than $U(\rho, L_c)$, defined as $1/(1-\rho)+1/L_c$, since $L_c \leqslant C^*$. $U(\rho, L_c)$ in percentage is shown in the next column. Then, the number of bins required by the solution obtained from our algorithm is in the column under the label H. Then, H/L_c in the next column shows that non of the solutions obtained from our algorithm require more than 109% of the optimum number of bins.

Finally, we compare the performance of the algorithm against an alternative algorithm proposed by Garey et al. in [1], that we denote as 2DF. 2DF is a straight forward adaptation of the algorithm known as *First Fit Decreasing* (FFD) for the usual bin packing problem. So, it basically carries out exactly the same procedure as FFD except that the items are handled in decreasing order of the maximum of x_j and y_j of each item. The number of bins required by 2DF is listed in the

Table 1
Summary of computational results

Case	Max x_j	Max y_j	ρ	L_C	$U(\rho, L_c)$ (%)	H	H/L_C (%)	2DF	H/2DF (%)
1	0.3193	0.2974	0.3193	61	149	57	107	62	98
2	0.3190	0.3240	0.3240	61	150	57	107	62	98
3	0.3023	0.3240	0.3240	62	150	57	109	62	100
4	0.3236	0.2986	0.3236	61	150	57	107	62	98
5	0.3277	0.3243	0.3277	63	150	58	109	64	98
6	0.3200	0.2933	0.3200	60	149	57	105	61	98
7	0.3168	0.3240	0.3240	62	150	58	107	63	98
8	0.3093	0.3240	0.3240	62	150	58	107	63	98
9	0.3208	0.3243	0.3243	62	150	57	109	62	100
10	0.3203	0.3243	0.3243	61	150	57	107	61	100

column under the label 2DF and the last column with label H/2DF shows that our algorithm performs slightly better than 2DF.

References

[1] Garey MR, Graham RL, Johnson DS, Yao AC. Resource constrained scheduling as generalized bin packing. Journal of Combinatorial Theory (A) 1976;21:257–98.
[2] Yao ACC. New algorithms for bin packing. Journal of the Association for Computing Machinery 1980;27:207–27.
[3] Fernandez de la Vega W, Lueker GS. Bin packing can be solved within $1+\varepsilon$ in linear time. Combinatorics 1981;1:349–55.
[4] Chekuri C, Khana S. On multi-dimensional packing problems. Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99). Baltimore, Maryland. 1999. 185–94.
[5] Spieksma FCR. A branch-and-bound algorithm for the two-dimensional vector packing problem. Computers Operations Research 1994;21:19–25.
[6] Woeginger GJ. There is no asymptotic PTAS for two-dimensional vector packing. Information Processing Letters 1997;64:293–7.
[7] Caprara A, Toth P. Lower bounds and algorithms for the 2-dimensional vector packing problem. Discrete Applied Mathematics 2001;111:231–62.
[8] Kellerer H, Kotov V. An approximation algorithm with absolute worst-case performance ratio 2 for two-dimensional vector packing. Operational Research Letters 2003;31:35–41.