Solving the Maximum Cardinality Bin Packing Problem with a Weight Annealing-Based Algorithm

Kok-Hua Loh[1], Bruce Golden[2], and Edward Wasil[3]

[1] School of Mechanical & Aerospace Engineering
Nanyang Technological University
Singapore 639798
khloh@ntu.edu.sg

[2] Robert H. Smith School of Business
Department of Decision and Information Technologies
University of Maryland
College Park, MD 20742
bgolden@rhsmith.umd.edu

[3] Kogod School of Business
American University
Washington, DC 20016
ewasil@american.edu

Summary. In the maximum cardinality bin packing problem (MCBPP), we have n items with different sizes and m bins with the same capacity. We want to assign a maximum number of items to the fixed number of bins without violating the capacity constraint on each bin. We develop a heuristic algorithm for solving the MCBPP that is based on weight annealing. Weight annealing is a metaheuristic that has been recently proposed in the physics literature. We apply our algorithm to two data sets containing 4,500 randomly generated instances and show that it outperforms an enumeration algorithm and a branch-and-price algorithm.

Key words: Bin packing; weight annealing; heuristics; combinatorial optimization.

1 Introduction

In the maximum cardinality bin packing problem, we are given n items with sizes t_i, $i \in N = \{1, \ldots, n\}$, and m bins of identical capacity c. The objective is to assign a maximum number of items to the fixed number of bins without violating the capacity constraint. The problem formulation is given by

$$\text{maximize} \quad z = \sum_{i=1}^{n} \sum_{j=1}^{m} x_{ij} \tag{1}$$

J.W. Chinneck et al. (eds.), *Operations Research and Cyber-Infrastructure*, Operations Research/Computer Science Interfaces Series 47, DOI: 10.1007/978-0-387-88843-9_8,

subject to

$$\begin{aligned} \sum_{i=1}^{n} t_i x_{ij} &\le c && j \in \{1,\ldots,m\} \\ \sum_{j=1}^{m} x_{ij} &\le 1 && i \in \{1,\ldots,n\} \\ x_{ij} &= 0 \text{ or } 1 && i \in \{1,\ldots,n\},\ j \in \{1,\ldots,m\} \end{aligned}$$

where $x_{ij} = 1$ if item i is assigned to bin j and $x_{ij} = 0$ otherwise.

The MCBPP is NP-hard (Labbé, Laporte, and Martello 2003). It has been applied in computing where we need to assign variable-length records to storage. The objective is to maximize the number of records stored in fast memory so as to ensure a minimum access time to the records given a fixed amount of storage space (Labbé, Laporte, and Martello 2003).

The MCBPP has been applied to the management of real-time multi-processors where the objective is to maximize the number of completed tasks with varying job durations before a given deadline (Coffman, Leung, and Ting 1978). It has been used to design processors for mainframe computers and the layout of electronic circuits (Ferreira, Martin, and Weismantel 1996).

A variety of bounds and heuristics have been developed for the MCBPP. Coffman, Leung, and Ting (1978) and Bruno and Downey (1985) provided probabilistic lower bounds. Kellerer (1999) considered this problem as a special case of the multiple knapsack problem where all items have the same profit and all knapsacks (or bins) have the same capacity and solved it with a polynomial approximation scheme for the multiple knapsack problem. Labbé, Laporte, and Martello (2003) developed several upper bounds and embedded them in an enumeration algorithm. Peeters and Degraeve (2006) solved the problem with a branch-and-price algorithm.

In this paper, we develop a heuristic algorithm for solving the MCBPP that is based on the concept of weight annealing. In Section 2, we describe weight annealing. In Section 3, we give the upper bounds and lower bounds that are used in our algorithm. In Section 4, we present our weight annealing algorithm. In Section 5, we apply our algorithm to 4,500 instances and compare our results to those produced by an enumeration algorithm and a branch-and-price algorithm. In Section 6, we summarize our contributions.

2 Weight Annealing

Ninio and Schneider (2005) proposed a weight annealing method that allowed a greedy heuristic to escape from a poor local optimum by changing the problem landscape and making use of the history of each optimization run. The authors changed the landscape by assigning weights to different parts of the solution space. Ninio and Schneider provided the following outline of their weight annealing algorithm.

Step 1. Start with an initial configuration from a greedy heuristic solution using the original problem landscape.
Step 2. Determine a new set of weights based on the previous optimization run and insight into the problem.
Step 3. Perform a new run of the greedy heuristic using the new weights.
Step 4. Return to Step 2 until a stopping criterion is met.

In their implementation, Ninio and Schneider required nonnegative values for all of the weights so their algorithm could look for good solutions. They used a cooling schedule with temperature T to change the values of the weights. When the value of T was large, there were significant changes to the weights. As T decreased, all weights approached a value of one. Ninio and Schneider applied their weight annealing to five benchmark traveling salesman problems with 127 to 1,379 nodes and generated results that were competitive with simulated annealing.

Weight annealing shares features with metaheuristics such as simulated annealing (e.g., a cooling schedule) and deterministic annealing (e.g., deteriorating moves) and similarities among these metaheuristics were presented by Ninio and Schneider (2005). In contrast to simulated annealing and deterministic annealing, weight annealing not only considers the value of the objective function, at each stage of an optimization run it also makes use of information on how well every part of the search space is being solved. By creating distortions in different parts of the search space (the size of the distortion is controlled by weight assignments based on insights gained from one iteration to the next), weight annealing seeks to expand and speed up the neighborhood search and focus computational efforts on the poorly solved regions of the search space.

3 Upper and Lower Bounds

3.1 Upper Bounds

Our algorithm uses upper bounds on the optimal value of the objective function (z^*) in (1) that were developed by Labbé, Laporte, and Martello (2003). The objective function value in (1) gives the maximum number of items that can be packed into the bins without violating bin capacities. Without loss of generality, we assume that the problem data are integers and $1 \leq t_1 \leq t_2 \leq \ldots \leq t_n \leq c$ (we refer to this as the ordered list throughout the rest of this paper).

The first upper bound for z^* developed by Labbé, Laporte, and Martello (2003) is given by

$$\bar{U}_0 = \max_{1 \leq k \leq n} \left\{ k \,:\, \sum_{i=1}^{k} t_i \leq mc \right\}. \tag{2}$$

Since the optimal solution is obtained by selecting the first z^* smallest items, all items with sizes t_i for which $i > \bar{U}_0$ can be disregarded.

Labbé, Laporte, and Martello (2003) derived the second upper bound $\bar{U}_1$ as follows. Let $Q(j)$ be the upper bound on the number of items that can be assigned to j bins. Then

$$Q(j) = \max \left\{ k \ : j \le k \le n, \sum_{i=1}^{k} t_i \le jc \right\} \text{ for } j = 1, \ldots, m. \tag{3}$$

An upper bound on z^* is given by

$$U_1(j) = Q(j) + \lfloor Q(j)/j \rfloor (m - j) \tag{4}$$

since $\lfloor Q(j)/j \rfloor$ is an upper bound on the number of items that can be packed into each of the remaining $(m - j)$ bins. The upper bound is obtained by taking the minimum over all j, that is,

$$\bar{U}_1 = \min_{j=1,\ldots,m} U_1(j). \tag{5}$$

Note that $\bar{U}_1$ dominates $\bar{U}_0$.

The third upper bound $\bar{U}_2$ from Labbé, Laporte, and Martello (2003) is derived in the following way. Let i be the smallest item in an instance with m bins. Then $m\lfloor c/t_1 \rfloor$ is an upper bound on the number of items that can be assigned to m bins because $\lfloor c/t_1 \rfloor$ is an upper bound on the number of items that can be packed into one bin. A valid upper bound is given by

$$\bar{U}_2(i) = (i - 1) + m\lfloor c/t_1 \rfloor. \tag{6}$$

If i is not the smallest item, then an optimal solution will contain all items $j < i$, and by taking the minimum over all i, we obtain a valid upper bound

$$\bar{U}_2 = \min_{j=1,\ldots,n} U_2(i). \tag{7}$$

It follows that the best a priori upper bound is given by $U^* = min\{\bar{U}_0, \bar{U}_1, \bar{U}_2\}$ (which is similar to what is given in Labbé, Laporte, and Martello (2003)). Since the optimal solution is obtained by selecting the first z^* smallest items, all items with sizes t_i for which $i > U^*$ can be disregarded. We point out that the time complexities for the computation of the bounds are given in the paper by Labbé, Laporte, and Martello (2003).

3.2 Lower Bounds

Our algorithm uses lower bounds developed by Martello and Toth (1990). Let I denote a one-dimensional bin packing problem instance. The lower bound L_2 on the optimal number of bins $z(I)$ can be computed in the following way.

Given any integer $\alpha,\ 0 \le \alpha \le c/2$, let

$$\begin{aligned} J_1 &= \{j \in N \ : \ t_j > c - \alpha\}, \\ J_2 &= \{j \in N \ : \ c - \alpha \ge t_j > c/2\}, \\ J_3 &= \{j \in N \ : \ c/2 \ge t_j \ge \alpha\}, \qquad N = \{1, \ldots, n\}, \end{aligned}$$

then

$$L(\alpha) = |J_1| + |J_2| + max\left(0, \left\lceil \frac{\sum_{j\in J_3} t_j - \left(|J_2|c - \sum_{j\in J_2} t_j\right)}{c} \right\rceil\right) \tag{8}$$

is a lower bound on $z(I)$.

L_2 is calculated by taking the maximum over α, that is,

$$L_2 = \max\{L(\alpha) \ : \ 0 \le \alpha \le c/2, \ \alpha \text{ integer}\} \tag{9}$$

In our algorithm, we use the Martello-Toth reduction procedure (denoted by MTRP and given in Martello and Toth 1990) to determine the lower bound L_3 which dominates L_2.

Let I be the original instance, z_1^r be the number of bins reduced after the first application of MTRP to I, and $I(z_1^r)$ be the corresponding residual instance. If $I(z_1^r)$ is relaxed by removing its smallest item, then we can obtain a lower bound by applying L_2 to $I(z_1^r)$ and this yields $L_1' = z_1^r + L_2(I(z_1^r)) \ge L_2(I)$. This process iterates until the residual instance is empty. For iteration k, we have a lower bound $L_k' = z_1^r + z_2^r + \ldots + z_k^r + L_2(I(z_k^r))$. Then

$$L_3 = \max\{L_1', L_2', \ldots, L_{k_{max}}'\} \tag{10}$$

is a valid lower bound for I where k_{max} is the number of iterations needed to have the residual instance empty.

4 Weight Annealing Algorithm for the MCBPP

In this section, we present our weight annealing algorithm for the maximum cardinality bin packing problem which we denote by WAMC. Table 1 illustrates WAMC in pseudo code. We point out that a problem has been solved to optimality once we have found a feasible bin packing for the current instance defined by the theoretical upper bound U^* at Step 4 of our algorithm.

The number of items (n), the ordered list of item sizes, the bin capacity (c), and the number of bins (m) are inputs. For the ordered list, the data are integers and $1 \le t_1 \le t_2 \le \ldots \le t_n \le c$, where t_i is the size of item i.

4.1 Computing the Bounds

We begin by computing the three upper bounds and then setting $U^* = min\{\bar{U}_0, \bar{U}_1, \bar{U}_2\}$. Since the optimal solution of any instance is obtained by selecting the first z^* smallest items, we update the ordered list by removing any item i with size t_i for which $i > U^*$.

To improve the upper bound, we compute L_3 by applying MTRP. If L_3 is greater than m, it is not feasible to pack the items on the ordered list into m bins, so we can reduce U^* by 1. We update the ordered list by removing any item i with size t_i for which $i > U^*$. We iterate until $L_3 = m$.

Table 1. Weight annealing algorithm (WAMC) for the MCBPP

Step 0. Initialization
  Parameters are K (scaling parameter), *nloop1, nloop2, T* (temperature), and *Tred*.
  Set $K = 0.05$, $nloop1 = 20$, $nloop2 = 50$, $T = 1$, and $Tred = 0.95$.
  Inputs are number of items (n), the item size ordered list, bin capacity (c), and number of bins (m).
Step 1. Compute the upper bound $U^* = \{\bar{U}_0, \bar{U}_1, \bar{U}_2\}$.
Step 2. Set $n = U^*$.
  Remove item $i > U^*$ from the ordered list.
Step 3. Improve the upper bound.
  While $(L_3 > m)$ do{
    $U^* = U^* - 1$.
    Remove item $i > U^*$ from the ordered list.
    Compute L_3.}
Step 4. For $j = 1$ to *nloop1*
  Step 4.1 Construct initial solution with the ordered list with modified FFD algorithm.
  Step 4.2 Improve the current solution.
    Set $T = 1$.
    Compute residual capacity r_i of bin i.
    For $k = 1$ to *nloop2*
      Compute weights $w_i^T = (1 + Kr_i)^T$.
    Do for all pairs of bins{
      Perform Swap (1,0)
        Evaluate feasibility and $\Delta f_{(1,0)}$.
        If $\Delta f_{(1,0)} \geq 0$
          Move the item.
          Exit Swap (1,0) and,
          Exit j loop and k loop if m is reached.
      Exit Swap (1,0) if no feasible move with $\Delta f_{(1,0)} \geq 0$ is found.
      Perform Swap (1,1)
        Evaluate feasibility and $\Delta f_{(1,1)}$.
        If $\Delta f_{(1,1)} \geq 0$
          Swap the items.
          Exit Swap (1,1) and,
          Exit j loop and k loop if m is reached.
      Exit Swap (1,1) if no feasible move with $\Delta f_{(1,1)} \geq 0$ is found.
      Perform Swap (1,2)
        Evaluate feasibility and $\Delta f_{(1,2)}$.
        If $\Delta f_{(1,2)} \geq 0$
          Swap the items.
          Exit Swap (1,2) and,
          Exit j loop and k loop if m is reached.
      Exit Swap (1,2) if no feasible move with $\Delta f_{(1,2)} \geq 0$ is found.
      Perform Swap (2,2)
        Evaluate feasibility and $\Delta f_{(2,2)}$.
        If $\Delta f_{(2,2)} \geq 0$
          Swap the items.
          Exit Swap (2,2) and,
          Exit j loop and k loop if m is reached.
      Exit Swap (2,2) if no feasible move with $\Delta f_{(2,2)} \geq 0$ is found.}
    $T := T \times Tred$
    End of k loop
  End of j loop
Step 5. Outputs are the number of bins used and the final distribution of items.

4.2 Weight Annealing for the Bin Packing Problem

Next, we solve the one-dimensional bin packing problem with the current ordered list. We start with an initial solution generated by the first-fit decreasing procedure (FFD) that we have modified in the following way. We select an item for packing with probability 0.5. In other words, we start with the first item on the ordered list and, based on a coin toss, we pack it into a bin if it is selected, or leave it on the ordered list if it is not selected. We continue down the ordered list until an item is selected for packing. We then pack the second item in the same manner and so on, until we reach the bottom of the list. For each bin i in the FFD solution, we compute the bin load l_i which is the sum of sizes of items in bin i (that is, $l_i = \sum_{j=1}^{q_i} t_{ij}$, where t_{ij} is the size of item j in bin i and q_i is the number of items in bin i), and the residual capacity r_i which is given by $r_i = (c - l_i)/c$.

4.2.1 Objective Function

In conducting our neighborhood search, we use the objective function given by Fleszar and Hindi (2002):

$$\text{maximize } f = \sum_{i=1}^{p} (l_j)^2 \tag{11}$$

where p is the number of bins in the current solution. This objective function seeks to reduce the number of bins along with maximizing the sum of the squared bin loads.

4.2.2 Weight Assignment

A key feature of our procedure is the distortion of item sizes that allows for both uphill and downhill moves. The changes in the apparent sizes of the items are achieved by assigning different weights to the bins and their items according to how well the bins are packed.

For each bin i, we assign weight ${w_i}^T$ according to

$${w_i}^T = (1 + Kr_i)^T \tag{12}$$

where K is a constant and T is a temperature parameter. We apply the weight to each item in the bin. The scaling parameter K controls the amount of size distortion for each item. T controls the amount by which a single weight can be varied. We start with a high temperature $(T = 1)$ and this allows more downhill moves. The temperature is reduced at the end of every iteration $(T \times 0.95)$, so that the amount of item distortion decreases and the problem space looks more like the original problem space.

At a given temperature T, the size distortion for an item is proportional to the residual capacity of its bin. At a local maximum, not-so-well packed bins will have

large residual capacities. We try to escape from a poor local maximum with downhill moves. To enable downhill moves, our weighting function increases the sizes of items in poorly packed bins.

Since the objective function tries to maximize the number of fully filled bins, the size transformation increases the chances of a swap between one of the enlarged items in this bin and a smaller item from another bin. Thus, we have an uphill move in the transformed space, which may be a downhill move in the original space. We make a swap as long as it is feasible in the original space.

4.2.3 Swap Schemes

We start the swapping process by comparing the items in the first bin with the items in the second bin, and so on, sequentially down to the last bin in the initial solution. Neighbors of a current solution can be obtained by swapping (exchanging) items between all possible pairs of bins. We use four different swapping schemes: Swap (1,0), Swap (1,1), Swap (1,2), and Swap (2,2). Fleszar and Hindi (2002) proposed the first two schemes.

In Swap (1,0), one item is moved from bin α to bin β. The change in the objective function value $(\Delta f_{(1,0)})$ that results from moving one item i with size $t_{\alpha i}$ from bin α to bin β is given by

$$\Delta f_{(1,0)} = (l_\alpha - t_{\alpha i})^2 + (l_\beta + t_{\alpha i})^2 - {l_\alpha}^2 - {l_\beta}^2. \tag{13}$$

In Swap (1,1), we swap item i from bin α with item j from bin β. The change in the objective function value that results from swapping item i with size $t_{\alpha i}$ from bin α with item j with size $t_{\beta j}$ from bin β is given by

$$\Delta f_{(1,1)} = (l_\alpha - t_{\alpha i} + t_{\beta j})^2 + (l_\beta - t_{\beta j} + t_{\alpha i})^2 - {l_\alpha}^2 - {l_\beta}^2. \tag{14}$$

In Swap (1,2), we swap item i from bin α with items j and k from bin β. The change in the objective function value that results from swapping item i with size $t_{\alpha i}$ from bin α with item j with size $t_{\beta j}$ and item k with size $t_{\beta k}$ from bin β is given by

$$\Delta f_{(1,2)} = (l_\alpha - t_{\alpha i} + t_{\beta j} + t_{\beta k})^2 + (l_\beta - t_{\beta j} - t_{\beta k} + t_{\alpha i})^2 - {l_\alpha}^2 - {l_\beta}^2. \tag{15}$$

In Swap (2,2), we swap item i and item j from bin α with item k and item l from bin β. The change in the objective function value that results from swapping item i with size $t_{\alpha i}$ and item j with size $t_{\alpha j}$ from bin α with item k with size $t_{\beta k}$ and item l with size $t_{\beta l}$ from bin β is given by

$$\Delta f_{(2,2)} = (l_\alpha - t_{\alpha i} - t_{\alpha j} + t_{\beta k} + t_{\beta l})^2 + (l_\beta - t_{\beta k} - t_{\beta l} + t_{\alpha i} + t_{\alpha j})^2 - {l_a}^2 - {l_\beta}^2. \tag{16}$$

For a current pair of bins (α, β), the swapping of items by Swap (1,0) is carried out as follows. The algorithm evaluates whether the first item (item i) in bin α can be moved to bin β without violating the capacity constraint of bin β in the original space. In other words, does bin β have enough original residual capacity to accommodate the original size of item i? If the answer is yes (the move is feasible), the

change in objective function value of the move in the transformed space is evaluated. If $\Delta f_{(1,0)} \geq 0$, item i is moved from bin α to bin β. After this move, if bin α is empty and the total number of utilized bins reaches the specified number of bins (m), the algorithm stops and outputs the final results. If bin α is still partially filled, or the lower bound has not been reached, the algorithm exits Swap (1,0) and proceeds to Swap (1,1). If the move of the first item is infeasible or $\Delta f_{(1,0)} < 0$, the second item in bin α is evaluated and so on, until a feasible move with $\Delta f_{(1,0)} \geq 0$ is found or all items in bin α have been considered and no feasible move with $\Delta f_{(1,0)} \geq 0$ has been found. The algorithm then performs Swap (1,1), followed by Swap (1,2), and Swap (2,2). In each of the swapping schemes, we always take the first feasible move with a nonnegative change in objective function value that we find.

We point out that the improvement step (Step 4.2) is carried out 50 times $(nloop2 = 50)$ starting with $T = 1$, followed by $T = 1 \times 0.95 = 0.95$, $T = 0.95 \times 0.95 = 0.9025$, etc. At the end of Step 4, if the total number of utilized bins has not reached m, we repeat Step 4 with another initial solution. We exit the program as soon as the required number of bins reaches m or after 20 runs $(nloop1 = 20)$.

5 Computational Experiments

We now describe the test instances, present results generated by WAMC, and compare WAMC's results to those reported in the literature.

5.1 Test Instances

In this section, we describe how we generated two sets of test instances,

5.1.1 Test Set 1

We followed the procedure described by Labbé, Laporte, and Martello (2003) to randomly generate the first set of test instances. Labbé et al. specified the values of three parameters: number of bins ($m = 2$, 3, 5, 10, 15, 20), capacity ($c = 100$, 120, 150, 200, 300, 400, 500, 600, 700, 800), and range of item size $[t_{min}, 99]$ ($t_{min} = 1$, 20, 50). For each of the 180 triples (m, c, t_{min}), we created 10 instances by generating item size t_i in an interval according to a discrete uniform distribution until the condition $\Sigma t_i > mc$ was met. This gave us a total of $180 \times 10 = 1,800$ instances which we denote by Test Set 1. We requested the 1,800 instances used by Labbé et al. (2003), but Martello (2006) replied that these instances were no longer available.

5.1.2 Test Set 2

Peeters and Degraeve (2006) extended the problems of Labbé et al. by multiplying the capacity c by a factor of 10 and enlarging the range of item size to $[t_{min}, 999]$. Rather than fixing the number of bins, Peeters and Degraeve fixed the expected number of generated items (denoted by $E(n')$). $E(n')$ is not an input for generating the

instances; it is implicitly determined by the number of bins and capacity. Since the item sizes are uniformly distributed on the interval $[t_{min}, 999]$, the expected item size is $(t_{min}+999)/2$ and $E(n') = 2cm/(t_{min}+999)$. Given the number of expected items $\bar{n}$ as an input, the number of bins m must be $\bar{n}(t_{min}+999)/2c$.

We randomly generated the second set of test instances with parameter values specified by Peeters and Degraeve: desired number of items ($\bar{n} = 100, 150, 200, 250, 300, 350, 400, 450, 500$), capacity ($c = 1000, 1200, 1500, 2000, 3000, 4000, 5000, 6000, 7000, 8000$), and range of item size $[t_{min}, 999]$ ($t_{min} = 1, 200, 500$). For each of the 270 triples $(\bar{n}, c, t_{min})$, we created 10 instances. This gave us a total of $270 \times 10 = 2,700$ instances which we denote by Test Set 2.

5.2 Computational Results

We coded WAMC in C and C++ and used a 3 GHz Pentium 4 computer with 256 MB of RAM. In the next two sections, we provide the results generated by WAMC on the two sets of test instances.

5.2.1 Results on Test Set 1

In Table 2, we show the average number of items (n) generated over 10 instances for each triple (m, c, t_{min}) in Test Set 1. In Table 3, we give the number of instances solved to optimality by WAMC. In Table 4, we give the average running time in seconds for WAMC.

Table 2. Average value of n over 10 instances for each triple (m, c, t_{min}) in Test Set 1

t_{min}	m	c									
		100	120	150	200	300	400	500	600	700	800
1	2	4	5	6	9	12	16	21	25	28	34
1	3	6	7	10	12	18	25	31	36	45	49
1	5	11	12	15	20	30	40	53	61	70	79
1	10	20	24	31	41	62	82	99	120	137	160
1	15	30	36	46	62	94	117	150	179	210	239
1	20	41	49	60	85	119	158	198	240	278	317
20	2	4	4	5	7	10	14	17	21	24	27
20	3	5	6	8	11	15	21	26	30	36	42
20	5	8	10	13	16	26	34	42	52	60	67
20	10	17	21	25	34	51	70	84	100	116	134
20	15	26	30	39	51	79	101	125	149	177	203
20	20	34	41	51	70	101	134	167	201	236	267
50	2	3	3	4	5	8	11	13	16	19	22
50	3	4	5	6	8	12	16	21	24	28	33
50	5	7	8	10	13	20	27	34	41	48	54
50	10	14	16	20	27	41	54	68	81	93	107
50	15	20	24	30	40	61	81	101	121	140	161
50	20	27	32	40	54	82	107	134	160	188	215

Table 3. Number of instances solved to optimality by WAMC in Test Set 1

t_{min}	m	c									
		100	120	150	200	300	400	500	600	700	800
1	2	10	10	10	10	10	10	10	10	10	10
1	3	10	10	10	10	10	10	10	10	10	10
1	5	10	10	10	10	10	10	10	10	10	10
1	10	10	10	10	10	10	10	10	10	10	10
1	15	10	10	10	10	10	10	10	10	10	10
1	20	9	10	10	10	10	10	10	10	10	10
20	2	10	10	10	10	10	10	10	10	10	10
20	3	10	10	10	10	10	10	10	10	10	10
20	5	10	10	10	10	10	10	10	10	10	10
20	10	9	10	9	10	10	10	10	10	10	10
20	15	10	10	9	10	10	10	10	10	10	10
20	20	9	10	10	10	10	10	10	10	10	10
50	2	10	10	10	10	10	10	10	10	10	10
50	3	10	10	10	10	10	10	10	10	10	10
50	5	10	10	10	9	10	10	10	10	10	10
50	10	10	10	10	9	10	10	10	10	10	10
50	15	10	10	10	10	10	10	10	10	10	10
50	20	10	10	10	10	10	10	10	10	10	10

Table 4. Average computation time (s) for WAMC over 10 instances for each triple (m, c, t_{min}) in Test Set 1

t_{min}	m	c									
		100	120	150	200	300	400	500	600	700	800
1	2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1	3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1	5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01
1	10	0.01	0.01	0.00	0.00	0.00	0.01	0.02	0.02	0.04	0.04
1	15	0.01	0.01	0.00	0.00	0.00	0.03	0.07	0.12	0.18	0.30
1	20	0.03	0.07	0.00	0.01	0.03	0.09	0.23	0.41	0.54	1.05
20	2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20	3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20	5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20	10	0.00	0.00	0.01	0.00	0.00	0.00	0.01	0.02	0.04	0.08
20	15	0.01	0.00	0.00	0.00	0.00	0.02	0.04	0.09	0.18	0.04
20	20	0.01	0.03	0.02	0.01	0.02	0.05	0.11	0.03	0.49	0.87
50	2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
50	3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
50	5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
50	10	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.03	0.07
50	15	0.00	0.00	0.00	0.09	0.00	0.01	0.02	0.03	0.07	0.18
50	20	0.00	0.00	0.00	0.06	0.00	0.02	0.03	0.05	0.18	0.37

We see that WAMC found optimal solutions to 1,793 instances. On average, WAMC is very fast with most computation times less than 0.01 s and the longest average time about 1 s.

Labbé, Laporte, and Martello (2003) generated 1,800 instances and solved each instance using a four-step enumeration algorithm (which we denote by LLM) on a Digital VaxStation 3100 (a slow machine that is comparable to a PC486/33). We point out that our Test Set 1 and the 1,800 instances used by Labbé et al. are very similar (the average values of n that we give in Table 2 are nearly the same as those given by Labbé et al. (2003), but they are not exactly the same). In Table 5, we provide the number of instances solved to optimality by LLM. We see that LLM found optimal solutions to 1,759 instances. On average, LLM is fast with many computation times 0.01 s or less and the longest average time several hundred seconds or more.

Peeters and Degraeve (2006) followed the procedure of Labbé et al. (2003) and generated 1,800 instances. They solved each instance using a branch-and-price algorithm (denoted by BP) on a COMPAQ Armada 700M, 500 MHz Intel Pentium III computer with a time limit of 900 seconds. Peeters and Degraeve reported that BP solved 920 instances ("... for those types of instances where the average CPU is significantly different from 0 ...") to optimality. For these 920 instances, most of the computation times were less than 0.01 s. Although not listed in the paper explicitly, we believe that BP also solved the remaining 880 instances to optimality.

In summary, on three different sets of 1,800 instances generated using the specifications of Labbé et al. (2003), the number of optimal solutions found by BP, WAMC, and LLM were 1,800, 1,793, and 1,759, respectively.

Table 5. Number of instances solved to optimality by LLM reported in Labbé et al. (2003)

t_{min}	m	c									
		100	120	150	200	300	400	500	600	700	800
1	2	10	10	10	10	10	10	10	10	10	10
1	3	10	10	10	10	10	10	10	10	10	10
1	5	10	10	9	10	10	10	10	10	10	10
1	10	10	9	10	10	10	10	10	10	10	10
1	15	10	8	10	10	10	10	10	10	10	10
1	20	10	10	10	10	10	10	10	10	10	10
20	2	10	10	10	10	10	10	10	10	10	10
20	3	10	10	10	9	10	10	10	10	10	10
20	5	10	9	8	9	10	10	10	10	10	10
20	10	10	9	10	10	10	10	10	10	10	10
20	15	10	10	8	10	10	10	10	10	10	10
20	20	10	10	8	10	10	10	10	10	10	10
50	2	10	10	10	10	10	10	10	10	10	10
50	3	10	10	10	10	10	10	10	10	10	10
50	5	10	10	10	10	10	10	10	10	10	10
50	10	10	10	10	10	9	10	10	10	10	10
50	15	10	10	10	7	7	7	10	10	10	10
50	20	10	10	10	8	6	5	8	9	7	10

5.2.2 Results on Test Set 2

In Table 6, we show the average number of bins (n) over 10 instances for each triple $(\bar{n},\ c,\ t_{min})$ in Test Set 2. In Table 7, we give the number of instances from Test Set 2 solved to optimality by WAMC. When the number of instances solved to optimality is less than 10 for WAMC, the maximum deviation from the optimal solution in terms of the number of items is shown in parentheses. In Table 7, we also provide the results generated by BP as reported in Peeters and Degraeve (2006). BP solved 2,700 instances that are similar to, but not exactly the same as, the instances in Test Set 2.

We see that WAMC found optimal solutions to 2,665 instances (there are a total of 2,700 instances). BP found optimal solutions to 2,519 instances.

WAMC performed better on instances with large bin capacities and BP performed better on instances with small bin capacities. WAMC solved all 1,080

Table 6. Average number of bins (m) over 10 instances for each triple $(\bar{n},\ c,\ t_{min})$ in Test Set 2

$\bar{n}$	t_{min}	c									
		1000	1200	1500	2000	3000	4000	5000	6000	7000	8000
100	1	50	42	33	25	17	13	10	8	7	6
100	200	60	50	40	30	20	15	12	10	9	7
100	500	75	62	50	37	25	19	15	12	11	9
150	1	75	63	50	38	25	19	15	13	11	9
150	200	90	75	60	45	30	22	18	15	13	11
150	500	112	94	75	56	37	28	22	19	16	14
200	1	100	83	67	50	33	25	20	17	14	13
200	200	120	100	80	60	40	30	24	20	17	15
200	500	150	125	100	75	50	37	30	25	21	19
250	1	125	104	83	63	42	31	25	21	18	16
250	200	150	125	100	75	50	37	30	25	21	19
250	500	187	156	125	94	62	47	37	31	27	23
300	1	150	125	100	75	50	38	30	25	21	19
300	200	180	150	120	90	60	45	36	30	26	22
300	500	225	187	150	112	75	56	45	37	32	28
350	1	175	146	117	88	58	44	35	29	25	22
350	200	210	175	140	105	70	52	42	35	30	26
350	500	262	219	175	131	87	66	52	44	37	33
400	1	200	167	133	100	67	50	40	33	29	25
400	200	240	200	160	120	80	60	48	40	34	30
400	500	300	250	200	150	100	75	60	50	43	37
450	1	225	188	150	113	75	56	45	38	32	28
450	200	270	225	180	135	90	67	54	45	39	34
450	500	337	281	225	169	112	84	67	56	48	42
500	1	250	208	167	125	83	63	50	42	36	31
500	200	300	250	200	150	100	75	60	50	43	37
500	500	375	312	250	187	125	94	75	62	54	47

Table 7. Number of instances solved to optimality by WAMC in Test Set 2 and the number of instances solved to optimality by BP reported in Peeters and Degraeve (2006)

$\bar{n}$	t_{min}	c									
		1000		1200		1500		2000		3000	
		BP	WAMC	BP	WAMC	BP	WAMC	BP	WAMC	BP	WAMC
100	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	7(1)	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
150	1	10	10	10	9(1)	10	10	10	10	10	10
	200	10	10	10	10	10	9(1)	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
200	1	10	8(1)	10	8(1)	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	10	10	10	9(3)	10	10	10	10
250	1	10	9(1)	10	10	10	10	10	10	10	10
	200	10	10	10	9(1)	10	9(1)	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
300	1	10	9(1)	10	8(1)	10	10	10	10	10	10
	200	10	10	10	10	10	8(1)	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
350	1	10	9(1)	10	9(1)	10	10	10	10	10	10
	200	10	10	10	9(2)	10	10	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
400	1	10	8(1)	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	8(1)	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
450	1	10	9(1)	10	8(1)	10	10	10	10	10	10
	200	10	10	10	9(2)	10	9(1)	10	10	9	10
	500	10	10	10	10	10	7(4)	10	10	10	10
500	1	10	9(1)	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	9(1)	10	10	9	10
	500	10	10	10	10	10	10	10	10	10	10
	Total	270	261	270	259	270	255	270	270	268	270

( ) Maximum deviation from the optimal solution in terms of the number of items for WAMC.

Table 7. (continued)

$\bar{n}$	t_{min}	c									
		4000		5000		6000		7000		8000	
		BP	WAMC	BP	WAMC	BP	WAMC	BP	WAMC	BP	WAMC
100	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
150	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
200	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
250	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	10	10	10	10	10	10	10	10
300	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	1	10	3	10	3	10	10	10
350	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	0	10	0	10	0	10	9	10
400	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	10	10
	500	10	10	0	10	0	10	0	10	0	10
450	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	9	10
	500	10	10	0	10	0	10	0	10	0	10
500	1	10	10	10	10	10	10	10	10	10	10
	200	10	10	10	10	10	10	10	10	9	10
	500	7	10	0	10	0	10	0	10	0	10
	Total	267	270	221	270	223	270	223	270	237	270

instances with large bin capacities ($c = 5000, 6000, 7000, 8000$) to optimality, while BP solved 904 large-capacity instances to optimality. Over the 1,620 small-capacity bins ($c = 1000, 1200, 1500, 2000, 3000, 4000$), BP solved 1,615 instances to optimality, while WAMC solved 1,585 instances to optimality.

In Table 8, we show the average computation time in seconds for WAMC and BP for the instances solved to optimality. To illustrate, for the triple ($\bar{n} = 200$, $c = 1000$, $t_{min} = 1$), WAMC solved eight instances and averaged 0.1 s, while BP solved all 10 instances and averaged 0.5 s. We point out that for several triples (e.g., ($\bar{n} = 350$, $c = 5000$, $t_{min} = 500$)), BP did not solve any instance to optimality, so that no average computation time is provided in the table.

Over all 2,665 instances solved to optimality, WAMC had an average computation time of 0.20 s. Over all 2,519 instances solved to optimality, BP had an average

Table 8. Average computation time (s) for WAMC and BP on instances solved to optimality

$\bar{n}$	t_{min}	c									
		1000		1200		1500		2000		3000	
		BP	WAMC	BP	WAMC	BP	WAMC	BP	WAMC	BP	WAMC
100	1	0.1	0.2	0.1	0.9	0.3	0.3	0.0	0.0	0.0	0.0
	200	0.1	0.0	0.1	0.0	0.4	0.2(7)	0.1	0.0	0.0	0.0
	500	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.0	1.4	0.0
150	1	0.1	0.0	0.1	0.6(9)	2.3	0.1	0.0	0.1	0.0	0.1
	200	0.1	0.0	0.1	0.6	2.2	0.2(9)	1.7	0.1	0.0	0.1
	500	0.0	0.0	0.0	0.0	0.1	0.1	4.8	0.0	8.0	0.0
200	1	0.5	0.1(8)	3.0	0.6(8)	6.0	0.2	0.0	0.2	0.0	0.2
	200	0.2	0.0	0.3	0.3	6.1	8.9	3.3	0.2	0.0	0.2
	500	0.0	0.0	0.1	0.0	0.2	0.1(9)	6.1	0.0	24.3	0.1
250	1	1.7	0.1(9)	1.9	6.6	11.8	0.2	0.0	0.2	0.0	0.2
	200	0.3	0.0	0.8	0.1(9)	14.9	9.3(9)	15.5	0.3	0.0	0.2
	500	0.0	0.0	0.1	0.3	0.3	0.5	13.4	0.1	36.4	0.1
300	1	1.3	1.4(9)	7.2	3.4(8)	15.8	0.6	10.7	0.6	0.0	0.8
	200	0.5	0.0	0.6	0.9	27.4	5.0	43.8	0.6	39.8	0.5
	500	0.4	0.0	0.1	0.0	0.4	0.4	16.9	0.1	71.7	0.2
350	1	2.1	0.1(9)	29.3	5.4(9)	0.0	0.9	0.0	1.1	0.0	1.5
	200	0.6	0.1	1.4	0.1(9)	41.2	11.7	86.6	1.2	94.1	0.9
	500	0.0	0.0	0.1	0.1	0.6	0.8	27.7	0.1	117.9	0.3
400	1	3.3	0.1(8)	47.2	1.8	52.4	1.3	0.0	1.8	0.0	2.6
	200	0.8	0.1	1.6	0.0	68.7	6.3(8)	139.1	1.9	0.1	1.3
	500	0.0	0.0	0.2	0.1	0.8	2.5	39.5	0.2	165.0	0.4
450	1	8.2	5.2(9)	46.5	18.2(8)	128.6	1.8	35.2	2.3	0.0	3.3
	200	1.0	0.1	2.3	0.2(9)	88.8	4.2(9)	207.5	3.0	0.1(9)	1.7
	500	0.0	0.0	0.2	0.1	0.7	1.0(7)	50.7	0.3	237.4	0.6
500	1	13.2	0.2(9)	64.9	8.6	71.7	2.9	7.7	3.6	0.0	6.0
	200	1.6	0.1	1.7	0.1	127.6	5.2(9)	374.7	4.7	0.4(9)	2.3
	500	0.0	0.0	0.2	0.1	1.1	1.8	58.1	0.4	250.4	0.9

() When the number of instances solved to optimality is less than 10, the number solved to optimality is given in parentheses.

Table 8. (continued)

$\bar{n}$	t_{min}	c									
		4000		5000		6000		7000		8000	
		BP	WAMC	BP	WAMC	BP	WAMC	BP	WAMC	BP	WAMC
100	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	200	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0
	500	0.6	0.0	1.5	0.0	0.6	0.0	0.0	0.0	0.0	0.1
150	1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.8	0.0	0.1
	200	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.2
	500	13.9	0.1	10.9	0.1	10.3	0.1	0.3	0.1	0.7	0.2
200	1	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.2
	200	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.4	0.0	0.4
	500	47.8	0.1	41.2	0.1	23.9	0.1	0.8	0.2	9.7	0.3
250	1	0.0	0.2	0.0	9.2	0.0	0.1	0.0	0.1	0.0	0.1
	200	0.0	0.2	0.0	0.2	0.0	0.1	0.0	0.1	0.0	0.1
	500	98.8	0.1	268.6	0.2	204.0	0.1	50.4	0.1	6.3	0.1
300	1	0.0	1.2	0.0	1.2	0.0	1.1	0.0	1.1	0.0	1.3
	200	0.0	1.3	0.0	1.2	0.0	1.4	0.1	1.6	0.0	1.5
	500	239.6	0.4	326.6(1)	0.5	115.3(3)	0.6	16.7(3)	1.0	7.4	1.0
350	1	0.0	2.0	0.0	2.1	0.0	2.3	0.0	2.0	0.0	1.8
	200	0.0	2.2	0.0	2.2	0.0	2.3	0.0	2.0	0.0	2.8
	500	318.1	0.6	**(0)	0.7	**(0)	0.9	**(0)	1.1	99.6(9)	1.5
400	1	0.0	2.9	0.0	3.4	0.0	3.5	0.0	3.6	0.0	2.8
	200	0.0	3.2	0.0	3.3	0.0	4.6	0.0	4.4	0.0	3.6
	500	400.9	0.8	**(0)	1.0	**(0)	1.2	**(0)	1.6	**(0)	2.6
450	1	0.0	5.1	0.0	5.2	0.0	6.0	0.0	5.7	0.0	4.8
	200	0.1	5.0	0.1	4.8	0.0	5.8	0.0	6.8	0.0	6.3
	500	578.4	1.2	**(0)	1.3	**(0)	1.9	**(0)	2.7	**(0)	3.3
500	1	0.0	7.1	0.0	7.3	0.0	8.5	0.0	8.6	0.0	7.6
	200	0.1	6.8	0.2	7.4	0.0	8.2	0.0	9.9	0.0	8.9
	500	693.7	1.5	**(0)	2.0	**(0)	2.4	**(0)	3.1	**(0)	4.3

** BP did not solve any of the 10 instances to optimality.

computation time of 2.85 s. The Pentium III computer used by Peeters and Degraeve (2006) to run BP is much slower than the Pentium 4 computer that we used to run WAMC.

We point out that our weight annealing algorithm is a robust procedure that can be used to solve several variants of bin packing and knapsack problems such as the dual bin packing problem (see Loh (2006) for more details).

6 Conclusions

We developed a new algorithm (WAMC) to solve the maximum cardinality bin packing problem that is based on weight annealing. WAMC is easy to understand and easy to code.

WAMC produced high-quality solutions very quickly. Over 4,500 instances that we randomly generated, our algorithm solved 4,458 instances to optimality with an average computation time of a few tenths of a second. Clearly, WAMC is a promising approach that deserves further computational study.

References

1. J. Bruno, P. Downey. Probabilistic bounds for dual bin-packing, *Acta Informatica*, 22:333–345, 1985.
2. E. Coffman, J. Leung, D. Ting. Bin packing: Maximizing the number of pieces packed, *Acta Informatica*, 9:263–271, 1978.
3. C. Ferreira, A. Martin, R. Weismantel. Solving multiple knapsack problems by cutting planes, *SIAM Journal on Optimization*, 6:858–877, 1996.
4. K. Fleszar, K. Hindi. New heuristics for one-dimensional bin-packing, *Computers & Operations Research*, 29:821–839, 2002.
5. H. Kellerer. A polynomial time approximation scheme for the multiple knapsack problem, in *Randomization, Approximation, and Combinatorial Algorithms and Techniques* (D. Hochbaum, K. Jansen, J. Rolim, and A. Sinclair, editors), 51–62, Springer, Berlin, 1999.
6. M. Labbé, G. Laporte, S. Martello. Upper bounds and algorithms for the maximum cardinality bin packing problem, *European Journal of Operational Research*, 149: 490–498, 2003.
7. K-H. Loh. Weight annealing heuristics for solving bin packing and other combinatorial optimization problems: Concepts, algorithms, and computational results, Ph.D. dissertation, Robert H. Smith School of Business, University of Maryland, College Park, Maryland, 2006.
8. S. Martello, private communication, 2006.
9. S. Martello, P. Toth. Lower bounds and reduction procedures for the bin-packing problem, *Discrete Applied Mathematics*, 26:59–70, 1990.
10. M. Ninio, J. Schneider. Weight annealing, *Physica A*, 349:649–666, 2005.
11. M. Peeters, Z. Degraeve. Branch-and-price algorithms for the dual bin packing and maximum cardinality bin packing problem, *European Journal of Operational Research*, 170:416–439, 2006.