Available online at www.sciencedirect.com
ScienceDirect
Computers & Industrial Engineering 53 (2007) 123–136
computers & industrial engineering
www.elsevier.com/locate/dsw

A two-level search algorithm for 2D rectangular packing problem

Mao Chen *, Wenqi Huang

School of Computer Science, Huazhong University of Science and Technology, 430074 Wuhan, People's Republic of China

Received 19 August 2006; received in revised form 21 December 2006; accepted 11 April 2007
Available online 18 April 2007

Abstract

In this paper, we propose a two-level search algorithm to solve the two-dimensional rectangle packing problem. In our algorithm, the rectangles are placed into the container one by one and each rectangle should be packed at a position by a corner-occupying action so that it touches two items without overlapping other already packed rectangles. At the first level of our algorithm, a simple algorithm called A_0 selects and packs one rectangle according to the highest degree first rule at every iteration of packing. At the second level, A_0 is itself used to evaluate the benefit of a CCOA more globally. Computational results show that the resulted packing algorithm called A_1 produces high-density solutions within short running times.

Keywords: Rectangle packing problem; Heuristic; Greedy algorithm

1. Introduction

The two-dimensional (2D) rectangular packing problem is an optimization problem of allocating a set of rectangular items to larger containers with the objective of minimizing the waste. The 2D rectangular packing problem has been widely studied in recent decades, as it has numerous applications in the cutting and packing industry, e.g. wood, glass and cloth industries, newspapers paging, VLSI floor planning and so on, with different applications incorporating different constraints and objectives (Chan & Markov, 2004; Hifi & Ouafi, 1998; Lee & Sewell, 1999; Lesh, Marks, & Mcmahon, 2004; Lodi, Martello, & Monaci, 2002). The 2D rectangular packing problem belongs to a subset of classical cutting and packing problems and has been shown to be an NP hard problem (Hochbaum & Maass, 1985). Therefore, various approximation or heuristic algorithms have been proposed for approaching the problem.

In rectangle packing, one of the earliest approaches is the so-called bottom-left (BL) heuristic (Baker, Coffman, & Rivest, 1980). In BL, the rectangles to be placed are, starting from the top-right corner of the container, first slide vertically downward as far as possible, followed by sliding horizontally as far as possible to the left, so as to reduce the number of possible packing patterns. One improved variant of BL is the

* Corresponding author.
E-mail address: mchen_1@163.com (M. Chen).

0360-8352/$ - see front matter
doi:10.1016/j.cie.2007.04.007

bottom-left-fill (BLF) heuristic, in which the rectangles are placed directly into the lowest positions available and then left justified. Both BL and BLF are later used in meta-heuristic based algorithms. Hybrid algorithms combining genetic algorithm (GA) and deterministic methods were proposed by Jakobs (1996), Liu and Teng (1999), and Dagli and Poshyanonda (1997). An empirical investigation of the result of different combinations of simulated annealing (SA) and GA with various heuristics such as BL or BLF was given by Hopper and Turton (2001).

Inspired by enhancing some working experience of professional masons in their everyday work, a Less Flexibility First (LFF) packing principle was proposed in Tam et al. (1998); Wu, Huang, and Lau (2002) for solving the rectangular packing problem. Recently, by introducing a tightness measure to represent the degree of fitting between rectangles and placement locations, an enhanced version of LFF, called LFFT, was proposed in Wu, Chan, and SAGA (2005). More recently, a fast heuristic recursive (HR) algorithm based on divide-and-conquer and greedy strategies was proposed in Zhang, Kang, and Deng (2006), and an effective GA, called SPGAL, was proposed in Bortfeldt (2006) for the four subtypes of the 2D strip packing problem.

Based on the previous studies (Huang, Li, Akeb, & Li, 2005; Tam et al., 1998; Wu et al., 2002), we will propose a quite different class of efficient heuristic for the 2D rectangular packing problem in this paper. Different from the stochastic search methods where the researches are mainly focused on finding the efficient data structures for presenting packing results so that the search space and the processing time of the underling search engine can be minimized, our algorithm is a deterministic algorithm and no data structure is needed. Similar to LFF, our algorithm packs the rectangles into the container one by one and each rectangle should be packed at a position occupying an empty corner in the container. The computational results of our proposed algorithm is very encouraging in terms of both packing density and running time.

The rest of this paper is organized as follows: Section 2 gives the problem definition. Section 3 describes the details of the algorithm. Section 4 presents our experimental results for two sets of test instances. We draw our conclusions in Section 5.

2. Problem definition

We consider the following rectangular packing problem: given a rectangular empty container with fixed width and infinite height and a set of rectangles with various sizes, the rectangle packing problem is to pack each rectangle into the container such that no two rectangles overlap and the used height of the container is minimized. From this optimization problem, an associated decision problem can be formally stated as follows:

Given a rectangular board with given width W and given height H, and n rectangles with length l_i and width w_i, $1 \leqslant i \leqslant n$, take the origin of the two-dimensional Cartesian coordinate system at the bottom-left corner of the container (see Fig. 1). The aim of this problem is to determine if there exist a solution composed of n sets of

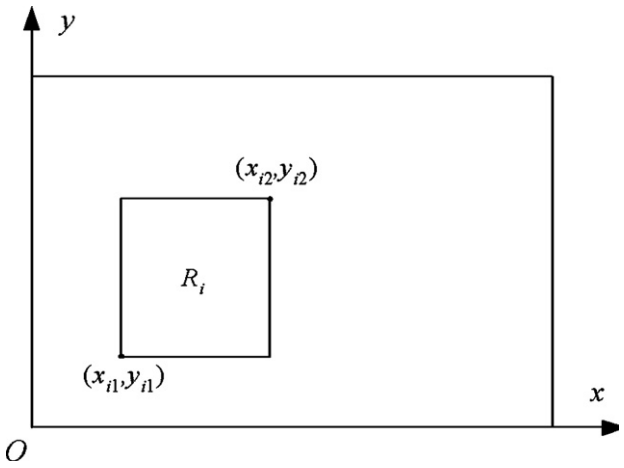

Fig. 1. Cartesian coordinate system.

quadruples $\{x_{11}, y_{11}, x_{12}, y_{12}\}, \ldots, \{x_{n1}, y_{n1}, x_{n2}, y_{n2}\}$, where (x_{i1}, y_{i1}) denotes the bottom-left corner coordinates of rectangle i, and (x_{i2}, y_{i2}) denotes the top-right corner coordinates of rectangle i. For all $1 \leqslant i \leqslant n$, the coordinates of rectangle i satisfy the following conditions:

(1) $x_{i2} - x_{i1} = l_i \wedge y_{i2} - y_{i1} = w_i$ or $x_{i2} - x_{i1} = w_i \wedge y_{i2} - y_{i1} = l_i$;
(2) For all $1 \leqslant i, j \leqslant n, j \neq i$, rectangle i and j cannot overlap, i.e., one of the following conditions should be met: $x_{i1} \geqslant x_{j2}$ or $x_{j1} \geqslant x_{i2}$ or $y_{i1} \geqslant y_{j2}$ or $y_{j1} \geqslant y_{i2}$;
(3) $0 \leqslant x_{i1}, x_{i2} \leqslant W$ and $0 \leqslant y_{i1}, y_{i2} \leqslant H$.

In our packing process, each rectangle is free to rotate and its orientation θ can be 0 (for "not rotated") or 1 (for "rotated by $\pi/2$"). It is noted that the orthogonal rectangular packing problems denote that the packing process has to ensure the edges of each rectangle are parallel to the x- and y-axis, respectively.

Obviously, if we can find an efficient algorithm to solve this decision problem, we can then solve the original optimization problem by using some search strategies. For example, we first apply dichotomous search to rapidly get a "good enough" upper bound for the height, then from this upper bound we gradually reduce it until the algorithm no longer finds a successful solution. The final upper bound is then taken as the minimal height of the container obtained by the algorithm. In the following discussion, we will only concentrate on the decision problem of fixed container.

3. Proposed algorithm

3.1. Corner-occupying action and degree

Definition (*Configuration*)**.** A configuration C is a pattern (layout) where m $(0 \leqslant m < n)$ rectangles have been already packed inside the container without overlap, and $n - m$ rectangles remain to be packed into the container.

A configuration is said to be successful if $m = n$, i.e., all the rectangles have been placed inside the container without overlapping. A configuration is said to be failure if $m < n$ and none of the rectangles outside the container can be packed into the container without overlapping. A configuration is said to be final if it is either a successful configuration or a failure configuration.

Definition (*Candidate corner-occupying action*)**.** Given a configuration with m rectangles packed, there may be many empty corners formed by the previously packed rectangles and the four sides of the container. Let rectangle i be the current rectangle to be packed, a candidate corner-occupying action (CCOA) is the placement of rectangle i at an empty corner in the container so that rectangle i touches the two items forming

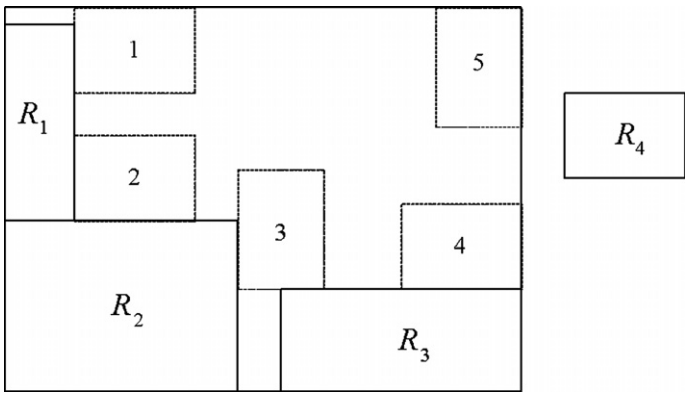

Fig. 2. Candidate corner-occupying action for rectangle R_4.

the corner and does not overlap other previously packed rectangles (an item may be a rectangle or one of the four sides of the container). Note that the two items are not necessarily touching each other.

Obviously, the rectangle to be packed has two possible orientation choices at each empty corner, that is, the rectangle can be placed with its longer side laid horizontally or vertically. A CCOA can be represented by a quadri-tuple (i, x, y, θ), where (x, y) is the coordinate of the bottom-left corner of the suggested location of rectangle i and θ is the corresponding orientation.

Under current configuration, there may be several candidate packing positions for the current rectangle to be packed. At the configuration in Fig. 2, three rectangles R_1, R_2 and R_3 are already placed in the container. In total, there are five empty corners to pack rectangle R_4, and R_4 can be packed at any one of them with two possible orientations. As a result, there are 10 CCOAs for R_4.

In order to prioritize the candidate packing choices, we need a concept that expresses the fitness value of a CCOA. Here, we introduce the quantified measure λ, called degree to evaluate the fitness value of a CCOA. Before presenting the definition of degree, we first introduce the definition of minimal distance between rectangles as follows:

Definition (*Minimal distance between rectangles*)**.** Let i and j be two rectangles already placed in the container, and (x_i, y_i), (x_j, y_j) are the coordinates of an arbitrary point on rectangle i and j, respectively. The minimal distance d_{ij} between i and j is:

$$d_{ij} = \min\left\{\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}\right\} \tag{1}$$

In Fig. 3, R_3 is packed on the position occupying the corner formed by the upper side and the right side of the container. As shown in Fig. 3, the minimal distance between R_3 and R_1, and the minimal distance between R_3 and R_2 are illustrated, respectively.

Definition (*Degree of CCOA*)**.** Let M be the set of rectangles already placed in the container. Rectangle i is the current rectangle to be packed, (i, x, y, θ) is one of the CCOAs for rectangle i. If corner-occupying action (i, x, y, θ) places rectangle i at a corner formed by two items (rectangle or side of the container) u and v, the degree λ of the corner-occupying action (i, x, y, θ) is defined as:

$$\lambda = 1 - d_{\min} \Big/ r\left(\frac{w_i + l_i}{2}\right) \tag{2}$$

where w_i and l_i are the width and the length of rectangle i, and $d_{\min}$ is the minimal distance from rectangle i to other rectangles in M and sides of the container (excluding u and v), that is,

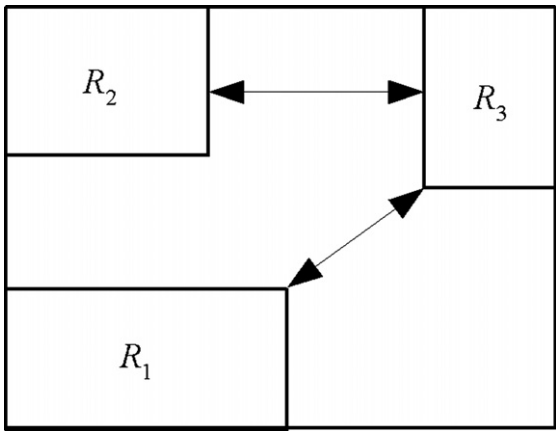

Fig. 3. Illustration of distance.

$$d_{\min} = \min\{d_{ij} | j \in M \cup \{s_1, s_2, s_3, s_4\}, j \neq u, v\} \quad (3)$$

where s_1, s_2, s_3 and s_4 are the four sides of the container.

It is clear that if a corner-occupying action places rectangle i at a position very close to the previously packed rectangles, the corresponding degree will be very high. Note that, if rectangle i can be packed by a CCOA at a corner in the container and touches more than two items, then $d_{\min} = 0$ and $\lambda = 1$; otherwise $\lambda < 1$. The degree of a corner-occupying action describes how the placed rectangle is close to the already existing pattern. Thus, we use it as the benefit of a packing step.

Intuitively, since one should place a rectangle as close as possible to the already existing pattern, it seems quite natural that the CCOA with the highest degree will be selected first to pack the rectangle into the container. We call this principle the highest degree first (HDF) rule.

3.2. The first level algorithm: A_0

Based on the HDF rule, our basic greedy packing algorithm called A_0 can be obtained directly. Algorithm A_0 is described as follows:

Procedure A_0 (C, L)
Begin
  While(L is not empty)
    For each CCOA in L
      Calculate the degree;
    Select the CCOA (i, x, y, θ) with the highest degree;
    Modify C by placing rectangle i at (x, y) with orientation θ;
    Modify L according to the new configuration C;
  Return C;
End.

At each iteration, a set of CCOAs for each of the unpacked rectangles is generated under current configuration C. Then the CCOAs for all the unpacked rectangles outside the container are gathered as a list L. A_0 calculates the degree of each CCOA in L and selects the CCOA (i, x, y, θ) with the highest degree λ, and place rectangle i at (x, y) with orientation θ. After placing rectangle i, the list L is modified as follows:

(1) Remove all the CCOAs involving rectangle i;
(2) Remove all infeasible CCOAs. A CCOA becomes infeasible because the involved rectangle would overlap rectangle i if it was placed;
(3) Re-calculate the degree λ of the remaining CCOAs;
(4) If a rectangle outside the container can be placed inside the container without overlap so that it touches rectangle i and a rectangle inside the container or the side of the container, create a new CCOA and put it into L, and compute the degree λ of the new CCOA.

If none of the rectangles outside the container can be packed into the container without overlap (L is empty) at certain iteration, A_0 stops with failure (returns a failure configuration). If all rectangles are packed in the container without overlap, A_0 stops with success (returns a successful configuration).

It should be pointed out that if there are several CCOAs with the same highest degree, we will select one that packs the corresponding rectangle closest to the bottom left corner of the container. For example, if CCOA_1 (i_1, x_1, y_1, θ_1) and CCOA_2 (i_2, x_2, y_2, θ_2) are of the same highest degree, we will select CCOA_1 if $(x_1 * x_1 + y_1 * y_1) < (x_2 * x_2 + y_2 * y_2)$. If there is still a tie, i.e., $(x_1 * x_1 + y_1 * y_1) = (x_2 * x_2 + y_2 * y_2)$, we will select CCOA_1 if CCOA_1 is before CCOA_2 in list L.

A_0 is a fast algorithm. However, given a configuration, A_0 only considers the relation between the rectangles already inside the container and the rectangle to be packed. It does not examine the relation between the

rectangles outside the container. In order to more globally evaluate the benefit of a CCOA and to overcome the limit of A_0, we compute the benefit of a CCOA using A_0 itself in the procedure *Benefit*A_1 to obtain our main packing algorithm called A_1.

3.3. The second level algorithm: A_1

Based on current configuration C, CCOAs for all unpacked rectangles are gathered as a list L. For each CCOA (i, x, y, θ) in L, the procedure *Benefit*A_1 is designed to evaluate its benefit more globally.

Procedure *Benefit*A_1 (i, x, y, θ, C, L)
Begin
  Let C' and L' be copies of C and L;
  Modify C' by placing rectangle i at (x, y) with orientation θ, and modify L';
  $C' = A_0\ (C', L')$;
  If (C' is a successful configuration)
  Return C';
    Else
  Return density (C');
End.

Given a copy C' of the current configuration C and a CCOA (i, x, y, θ) in L, *Benefit*A_1 begins by packing rectangle i in the container at (x, y) with orientation θ and call A_0 to reach a final configuration. If A_0 stops with success then *Benefit*A_1 returns a successful configuration, otherwise *Benefit*A_1 returns the density (the ratio of the total area of the rectangles inside the container to the area of the container) of a failure configuration as the benefit of the CCOA (i, x, y, θ). In this manner, *Benefit*A_1 evaluates all existing CCOAs in L.

Now, using the procedure *Benefit*A_1, the benefit of a CCOA is measured by the density of a failure configuration. The main algorithm A_1 is presented as follow:

Procedure A_1 ()
Begin
  Generate the initial configuration C;
  Generate the initial CCOA list L;
  While (L is not empty)
    maximum benefit $\leftarrow 0$
    For each CCOA (i, x, y, θ) in L
      $d =$ *Benefit*A_1 (i, x, y, θ, C, L);
      If (d is a successful configuration)
        Stop with success;
    Else
        Update the maximum benefit with d;
    Select the CCOA $(i^*, x^*, y^*, \theta^*)$ with the maximum benefit;
    Modify C by placing rectangle i^* at (x^*, y^*) with orientation θ^*;
    Modify L according to the new configuration C;
  Stop with failure
End.

Similarly, A_1 selects the CCOA with the maximum benefit and packs the corresponding rectangle into the container by this CCOA at each iteration. If there are several CCOAs with the maximum benefit, we select one that packs the corresponding rectangle closest to the bottom left corner of the container.

3.4. Computational complexity

We analyze the complexity of A_1 in the worst case, that is, when it cannot find a successful configuration, and discuss the real computational cost to find a successful configuration.

A_0 is clearly polynomial. Since every pair of rectangles or sides in the container can give a possible CCOA for a rectangle outside the container, the length of L is bounded by $O(m^2(n-m))$, if m rectangles are already placed in the container. For each CCOA in L, $d_{\min}$ is calculated using the $d_{\min}$ in the last iteration in O(1) time. The creation of new CCOAs and the calculation of their degree is also bounded by $O(m^2(n-m))$ since there are at most $O(m(n-m))$ new CCOAs (a rectangle might form a corner position with each rectangle in the container and each side of the container). So the time complexity of A_0 is bounded by $O(n^4)$.

A_1 uses a powerful search strategy in which the consequence of each CCOA is evaluated by applying *Benefit*A_1 in full, which allows us to examine the relation between all rectangles (inside and outside the container). Note that the benefit of a CCOA is measured by the density of a final configuration, which means that we should apply *Benefit*A_1 though to the end each time. At every iteration of A_1, *Benefit*A_1 uses a $O(n^4)$ procedure to evaluate all $O(m^2(n-m))$ CCOAs, therefore, the complexity of A_1 is bounded by $O(n^8)$.

It should be pointed out that the above upper bounds of the time complexity of A_0 and A_1 are just rough estimations, because most corner positions are infeasible to place any rectangle outside the container, and the real number of CCOAs in a configuration is thus much smaller than the theoretical upper bound $O(m^2(n-m))$.

The real computational cost of A_0 and A_1 to find a successful configuration is much smaller than the above upper bound. When a successful configuration is found, *Benefit*A_1 does not continue to try other CCOAs, nor A_1 to exhaust the search space. In fact, every call to A_0 in *Benefit*A_1 may lead to a successful configuration and then stops the execution at once. Then, the real computational cost of A_1 essentially depends on the real number of CCOAs in a configuration and the distribution of successful configurations. If the container height is not close to the optimal one, there exists many successful configurations, and A_1 can quickly find one. However, if the container height is very close to the optimal one, few successful configurations exist in the search space, and then A_1 may need to spend more time to find a successful configuration in this case.

4. Computational results

4.1. Computational results on rectangle packing problems

The first set of tests is done using the Hopper and Turton instances Hopper and Turton, 2001. There are 21 different sized test instances ranging from 16 to 197 items, and the optimal packing solutions of these test instances are all known (see Table 1). We implemented A_1 in C on a 2.4 GHz PC with 512 MB memory.

In order to find the minimal height, the experiments started with the height of the container set to be the optimal height. If A_1 could not find a successful configuration, the height of the container is increased by one unit at a time and the experiments are repeated until a successful configuration is obtained. As shown in Table 1, A_1 generates optimal solutions for 8 of the 21 instances; for the remaining 13 instances, the optimum is missed in each case by a single length unit.

To evaluate the performance of the algorithm, we compare A_1 with two of the best meta-heuristic (GA + BLF and SA + BLF) in Hopper and Turton (2001), HR Zhang et al. (2006), LFFT Wu et al. (2005) and SPGAL Bortfeldt (2006). In comparing different algorithms for solving rectangle packing problems, quality of solution and speed are the two most important measures of their performance. The quality of a solution is measured by the percentage gap, i.e., the relative distance of the solution lU to the optimum length lOpt. The gap is computed as (lU − lOpt)/lOpt. The indicated gaps for the seven classes are averaged over the respective three instances. As shown in Table 2, the gaps of A_1 ranges form 0.0% to 1.64% with the average gap 0.72, whereas the average gap of the two meta-heuristics and HR are 4.6%, 4.0% and 3.97%, respectively. Obviously, A_1 considerably outperforms these algorithms in terms of packing density. Compared with two other methods, the average gap of A_1 is lower than that of LFFT, however, the average gap of A_1 is slightly higher than that of SPGAL.

Table 1
Computational results of our algorithm for the test instances from Hopper and Turton instances Hopper and Turton, 2001

Test instance class/ subclass		No. of pieces	Object dimensions	Optimal height	Minimum height by A_1	% of unpacked area	CPU time (s)
C1	C11	16	20×20	20	20	0.00	0.37
	C12	17	20×20	20	20	0.00	0.50
	C13	16	20×20	20	20	0.00	0.23
C2	C21	25	15×40	15	15	0.00	0.59
	C22	25	15×40	15	15	0.00	0.44
	C23	25	15×40	15	15	0.00	0.79
C3	C31	28	30×60	30	30	0.00	3.67
	C32	29	30×60	30	30	0.00	1.44
	C33	28	30×60	30	31	3.23	0.03
C4	C41	49	60×60	60	61	1.64	0.22
	C42	49	60×60	60	61	1.64	0.13
	C43	49	60×60	60	61	1.64	0.11
C5	C51	73	90×60	90	91	1.09	0.34
	C52	73	90×60	90	91	1.09	0.33
	C53	73	90×60	90	91	1.09	0.52
C6	C61	97	120×80	120	121	0.83	8.73
	C62	97	120×80	120	121	0.83	0.73
	C63	97	120×80	120	121	0.83	2.49
C7	C71	196	240×160	240	241	0.41	51.73
	C72	197	240×160	240	241	0.41	37.53
	C73	196	240×160	240	241	0.41	45.81

Table 2
The gaps (%) and the running time (seconds) for meta-heuristics, HR, LFFT, SPGAL and A_1

Class	GA + BLF[a]		SA + BLF[a]		HR[b]		LFFT[c]		SPGAL[d]		A_1[e]	
	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time (s)	Gap	Time
C1	4.0	60	4.0	42	8.33	0	0.0	1	1.7	–	0.00	0.37
C2	7.0	120	6.0	144	4.45	0	0.0	1	0.0	–	0.00	0.61
C3	5.0	180	5.0	240	6.67	0.03	1.0	2	2.2	–	1.07	1.71
C4	3.0	780	3.0	1980	2.22	0.14	2.0	15	0.0	–	1.64	0.15
C5	4.0	2160	3.0	6900	1.85	0.69	1.0	31	0.0	–	1.09	0.40
C6	4.0	5160	3.0	22,920	2.5	2.21	1.0	92	0.3	–	0.83	3.98
C7	5.0	46,620	4.0	250,800	1.8	36.07	1.0	2150	0.3	–	0.41	45.02
Average gap (%)	4.6		4.0		3.97		0.86		0.64		0.72	

[a] PC with a Pentium Pro 200 MHz processor and 65 MB memory (Hopper and Turton, 2001).
[b] Dell GX260 with a 2.4 GHz CPU (Zhang et al., 2006).
[c] PC with a Pentium 4 1.8 GHz processor and 256 MB memory (Wu et al., 2005).
[d] The machine is 2 GHz Pentium (Bortfeldt, 2006).
[e] 2.4 GHz PC with 512 MB memory.

In order to compare the speed of the algorithms as fairly as possible, we first consulted the SPEC web page (http://www.specbench.org) for the speed of related CPUs, then we converted the running time of the algorithms according to the relative speed of their CPUs. The machines for the five algorithms are listed in Table 2. Our PC with a 2.4 GHz CPU is about 13 times faster than PC with a Pentium Pro 200 MHz CPU for the two meta-heuristics (GA + BLF, SA + BLF), therefore, we can convert the running time of the two meta-heuristics by dividing by 13. After converting all running times of the other four algorithms based on the relative speed of the CPUs, we can have a fair comparison of the speed. Obviously, the speed of A_1 is faster than that of GA + BLF, SA + BLF and LFFT. HR is a fast algorithm, whose time complexity is only $O(n^3)$ Zhang et al., 2006. For each class of instance, Table 2 shows that the average running time of A_1 is very similar to that of HR. Note that HR was tested

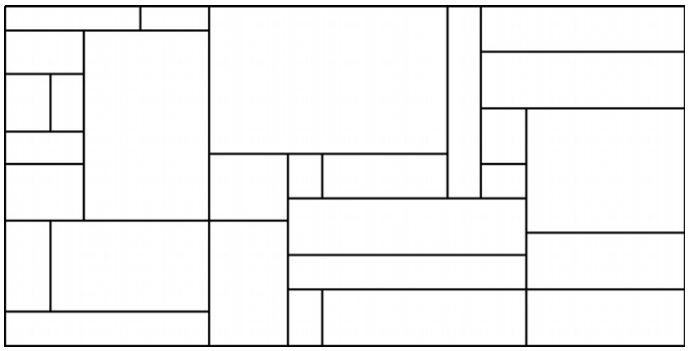

Fig. 4. Packing result of C31.

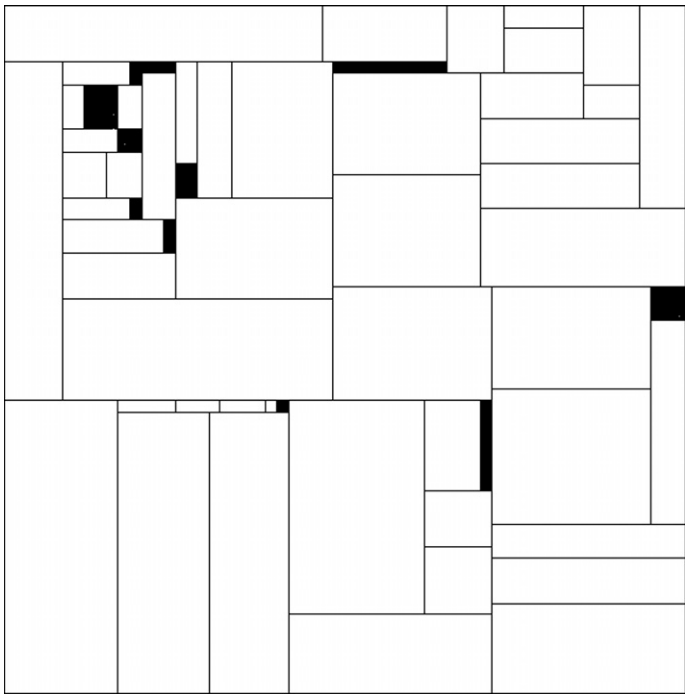

Fig. 5. Packing result of C42.

on Dell GX260 with a 2.4 GHz CPU, while A_1 was tested on a PC with 2.4 GHz CPU too. Unfortunately, the running time of each instance for SPGAL is not reported in the literature.

Table 2 shows that the running time of A_1 does not consistently correlate with its theoretical time complexity. For example, the average time of C3 is 1.71 s, while the average time of C4 and C5 are both within 0.5 s. As pointed out in the time complexity analysis, once A_0 finds a successful solution, the calculation of A_1 will terminate. Actually, the average time complexity is much smaller than the theoretical upper bound.

In addition, we give the packing results on test instances C31, C42 and C73 for A_1 in Figs. 4–6. Here, the packing result of C31 is of optimal height, and the heights for C42 and C73 are only one length unit higher than the optimal height, respectively.

4.2. Computational results on VLSI benchmark problems

To examine the efficiency of the proposed algorithm, we also apply A_1 to the MCNC benchmarks (*apte*, *xerox*, *hp*, *ami*33, *ami*49) and GSRC benchmarks (n100, n100b, n200, n200b, n300) Chan and Markov,

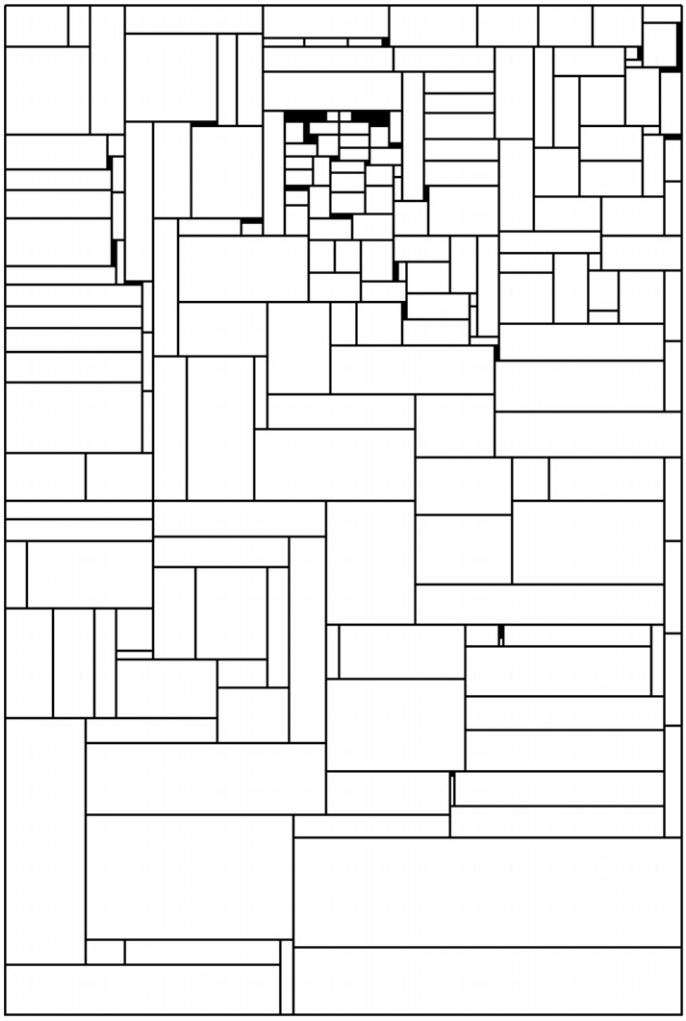

Fig. 6. Packing result of C73.

2004. These two sets of benchmarks were developed specially for building block layout (BBL) design and have been widely used for floorplanning and placement algorithm testing.

In our experiment, we consider the packed blocks as rigid rectangles whose shapes are not changeable. The goal is to find the minimum bounding rectangular container to pack in all the rectangles. The experiments started with the bounding square set to be the value close to the total area of the rectangles to be packed. If A_1 cannot produce a successful solution, we continue our experiments with the height or width of the container increased by one length unit at a time. The experiments were repeated until a successful solution was obtained.

The experimental results are presented in Table 3, compared with the results of BLOBB Chan and Markov (2004) and LFFT Wu et al. (2005). Based on multi-level branch-and-bound, BloBB is competitive with annealing-based algorithms in terms of runtime and solution quality.

It can be noted that A_1 can obtain results with area usage over 98% for all the test instances except for instance *xerox*, with the average packing density over 98.5%. For the five GSRC benchmarks with large size, the area usage is 98.57%, 98.45%, 98.64%, 98.65% and 98.75%, respectively. Obviously, compared with BloBB and LFFT, A_1 achieves higher area usage for most cases (only lower for instance n100 than LFFT). After considering the difference of the computational environment, the speeds of the three algorithms are similar.

Table 3
Packing results for MCNC and GSRC problem instances

Test case	No. of pieces	BloBB[a]			LFFT			A_1		
		Area	Density (%)	Time (s)	Area	Density (%)	Time (s)	Area	Density (%)	Time (s)
apte	9	47.30	98.44	0.04	–	–	–	46.92	99.23	0.01
xerox	10	20.31	95.27	0.08	–	–	–	19.81	97.69	0.01
hp	11	9.26	95.37	0.03	–	–	–	8.95	98.67	0.02
ami33	33	1.25	92.52	1.73	1.177	98.26	10	1.17	98.63	2.01
ami49	49	38.18	92.84	3.01	36.24	97.81	77	36.07	98.26	56.61
n100	100	192,234	93.38	5.62	179,776	99.84	1	182,104	98.57	8.22
n100b	100	175,263	91.36	34.7	–	–	–	162,328	98.45	6.16
n200	200	191,040	93.96	7.09	183,184	95.91	6	178,120	98.64	9.70
n200b	200	187,824	92.96	13.34	–	–	–	176,985	98.65	4.78
n300	300	297,018	91.97	11.04	283,024	96.52	17	276,640	98.75	37.23
Average packing density (%)			93.81			97.51			98.55	

[a] Best result from 10 runs, and the machine is a 1.2 GHz Linux Athlon workstation.

Figs. 7 and 8 illustrate the packing results of *ami*33 and *ami*49, and the area usage is 98.72% and 98.26%, respectively. Figs. 9 and 10 illustrate the packing a result of the two large GSRC benchmarks, n200 and n300, and the area usage is 98.64% and 98.75%, respectively.

5. Conclusions

Two-dimensional rectangle packing problems have applications in many industries and have been studied extensively. Many pure meta-heuristic algorithms (GA, SA, etc.) and mixed meta-heuristic algorithms (GA + SA, GA + BLF, etc.) have been proposed in past decades. Although some of them can obtain good results for test instances with small to medium size, they might not be practical for large problems since the running time is rather long.

A two-level search algorithm for the two-dimensional packing problem is presented in this paper. This algorithm is simple and intuitive, and the computational results have shown that our proposed algorithm

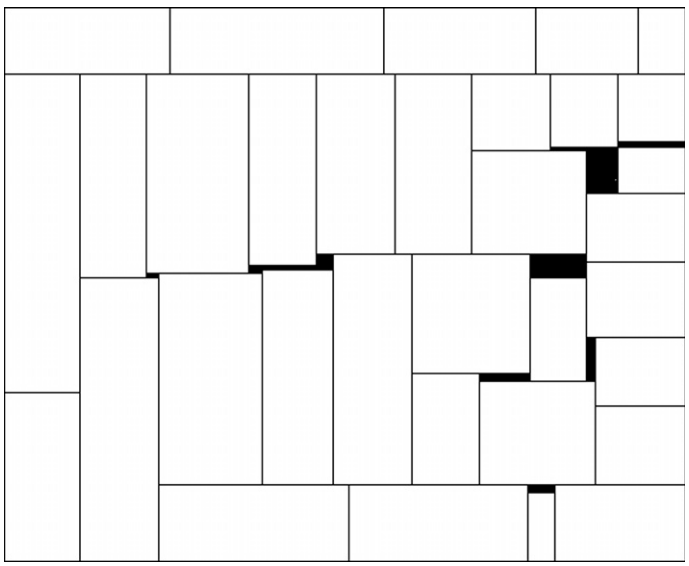

Fig. 7. Packing result of *ami*33.

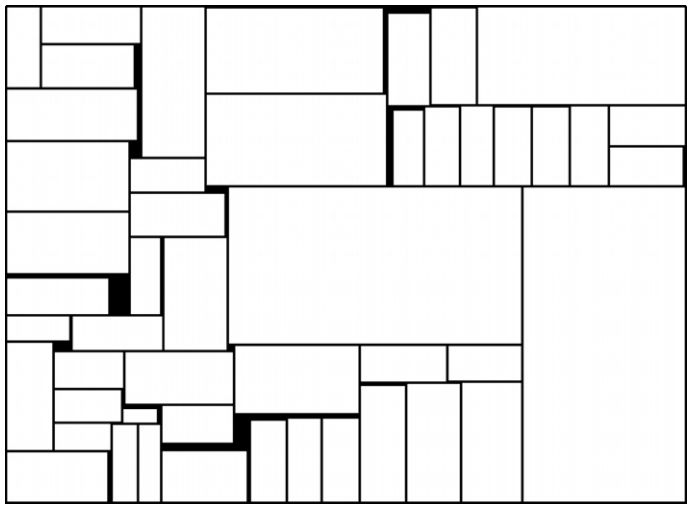

Fig. 8. Packing result of *ami*49.

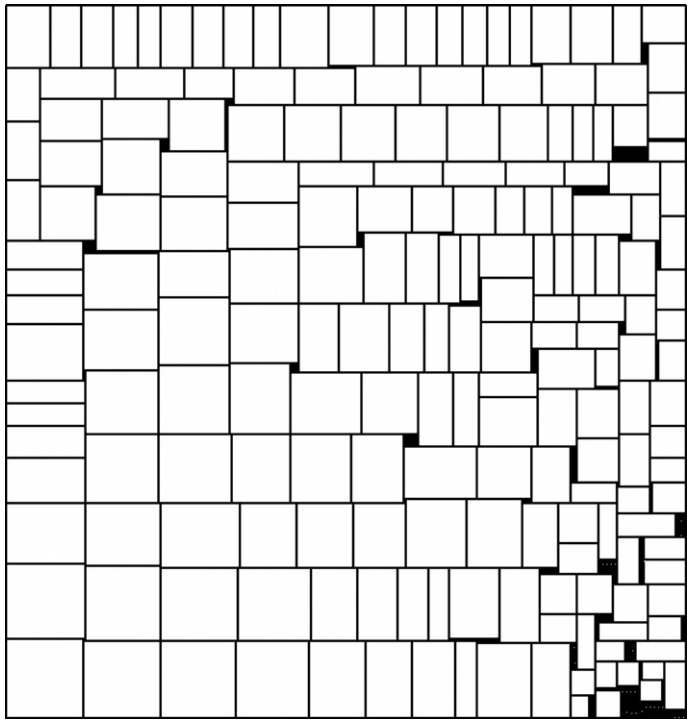

Fig. 9. Packing result of n200.

is quite effective: In Hopper and Turton stock cutting problems, our algorithm achieved over 99% average packing density in short running time, which is significantly better than the 96% packing density obtained by the known meta-heuristic algorithms; whereas in rectangle packing applying the well-known MCNC and GSRC benchmarks, we achieved much better packing density (over 98.5%) than two other effective algorithms and their running speeds are comparable. The encouraging results seem to be suggesting that our algorithm may be of great practical value to the rational layout of the rectangular objects in industrial fields.

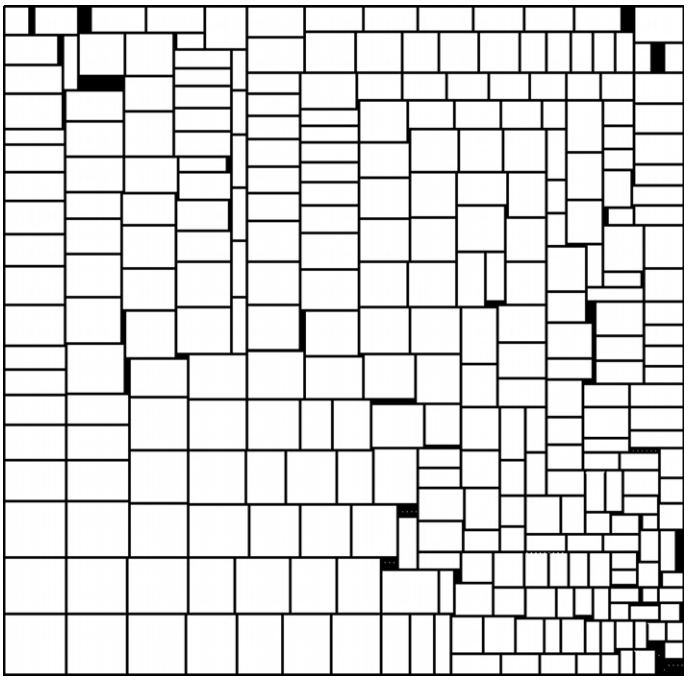

Fig. 10. Packing result of n300.

Acknowledgements

The authors to thank the referees for their helpful comments and suggestions that help to improve this paper. This work is supported by the National Grand Fundamental Research 973 Program of China (No. 2004CB318000) and the National Natural Science Foundation of China (No. 10471051).

References

Baker, B. S., Coffman, E. G., Jr., & Rivest, R. L. (1980). Orthogonal packing in two dimensions. *SIAM Journal on Computing, 9*, 846–855.

Bortfeldt, A. (2006). A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces. *European Journal of Operational Research, 172*, 814–837.

Chan, H. H., Markov, I. L. Practical slicing and non-slicing block-packing without simulated annealing. In: Proceedings of the great lakes symposium on VLSI (GLSVLSI), 2004, pp. 282–287.

Dagli, C. H., & Poshyanonda, P. (1997). New approaches to nesting rectangular patterns. *Journal of Intelligent Manufacturing, 8*, 177–190.

Hifi, M., & Ouafi, R. (1998). A best-first branch-and-bound algorithm for orthogonal rectangular packing problems. *International Transactions on Operational Research, 5*(5), 345–356.

Hochbaum, D. S., & Maass, W. (1985). Approximation schemes for covering and packing problems in image processing and VLSI. *Journal of the Association for Computing Machinery, 32*(1), 130–136.

Hopper, E., & Turton, B. (2001). An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem. *European Journal of Operational Research, 128*, 34–57.

Huang, W. Q., Li, Y., Akeb, H., & Li, C. M. (2005). Greedy algorithms for packing unequal circles into a rectangular container. *Journal of the Operational Research Society, 56*, 539–548.

Jakobs, S. (1996). On genetic algorithms for the packing of polygons. *European Journal of Operational Research, 88*, 165–181.

Lee, H. F., & Sewell, E. C. (1999). The strip-packing problem for a boat manufacturing firm. *IIE Transactions, 31*, 639–651.

Lesh, N., Marks, J., Mcmahon, A., et al. (2004). Exhaustive approaches to 2D rectangular perfect packing. *Information Processing Letters, 90*, 7–14.

Liu, D., & Teng, H. (1999). An improved BL-algorithm for genetic algorithm of the orthogonal packing of rectangles. *European Journal of Operational Research, 112*, 413–419.

Lodi, A., Martello, S., & Monaci, M. (2002). Two-dimensional packing problems: A survey. *European Journal of Operational Research, 141*, 241–252.

Tam, Y. O., Wu, Y. L., Huang, W. Q., Wong, C. K. An effective quasi-human based heuristic for solving rectangle packing problems, In: Proceedings of IEEE APCCAS: Microelectronics and integration system, Thailand, 1998, pp. 137–140.

Wu, Y. L., Chan, C. K. On improved least flexibility first heuristics superior for packing and stock cutting problems. In: Proceedings for stochastic algorithms: foundations and applications, SAGA 2005, Moscow, 2005, pp. 70–81.

Wu, Y. L., Huang, W. Q., Lau, S. C., et al. (2002). An effective quasi-human based heuristic for solving the rectangle packing problem. *European Journal of Operational Research, 141*, 341–358.

Zhang, D. F., Kang, Y., & Deng, A. S. (2006). A new heuristic recursive algorithm for the strip rectangular packing problem. *Computers & Operational Research, 33*, 2209–2217.