A Greedy Algorithm with Forward-Looking Strategy

Mao Chen
Engineering Research Center for Educational Information Technology,
Huazhong Normal University,
China

1. Introduction

The greedy method is a well-known technique for solving various problems so as to optimize (minimize or maximize) specific objective functions. As pointed by Dechter *et al* [1], greedy method is a controlled search strategy that selects the next state to achieve the largest possible improvement in the value of some measure which may or may not be the objective function. In recent years, many modern algorithms or heuristics have been introduced in the literature, and many types of improved greedy algorithms have been proposed. In fact, the core of many Meta-heuristic such as simulated annealing and genetic algorithms are based on greedy strategy.

"The one with maximum benefit from multiple choices is selected" is the basic idea of greedy method. A greedy method arrives at a solution by making a sequence of choices, each of which simply looks the best at the moment. We refer to the resulting algorithm by this principle the basic greedy (BG) algorithm, the details of which can be described as follow:

```
Procedure BG (partial solution $S$, sub-problem $P$)
Begin
    generate all candidate choices as list $L$ for current sub-problem $P$;
    while ($L$ is not empty OR other finish condition is not met)
        compute the fitness value of each choice in $L$;
        modify $S$ and $P$ by taking the choice with highest fitness value;
        update $L$ according to $S$ and $P$;
    end while;
    return the quality of the resulting complete solution;
End.
```

For an optimization problem, what remains is called a sub-problem after making one or several steps of greedy choice. For problem or sub-problem P, let S be the partial solution, and L be the list of candidate choices at the current moment.

To order or prioritize the choices, some evaluation criteria are used to express the fitness value. According to the BG algorithm, the candidate choice with the highest fitness value is selected, and the partial solution is updated accordingly. This procedure repeated step by step until a resulting complete solution is obtained.

Source: Advances in Greedy Algorithms, Book edited by: Witold Bednorz,
ISBN 978-953-7619-27-5, pp. 586, November 2008, I-Tech, Vienna, Austria

The representation of the BG algorithm can be illustrated by a search tree as shown in Fig.1. Each node in the search tree corresponds to a partial solution, and a line between two nodes represents the decision to add a candidate choice to the existing partial solution. Consequently, leaf nodes at the end of tree correspond to complete solutions.
In Fig.1, the black circle at level 1 denotes an initial partial solution. At level 2, there are four candidate choices for current partial solution, which denotes by four nodes. In order to select the best node, promise of each node should be determined. After some evaluation function has been employed, the second node with highest benefit (the circle in gray at level 2) is selected. Then, the partial solution and sub-problem are updated accordingly.

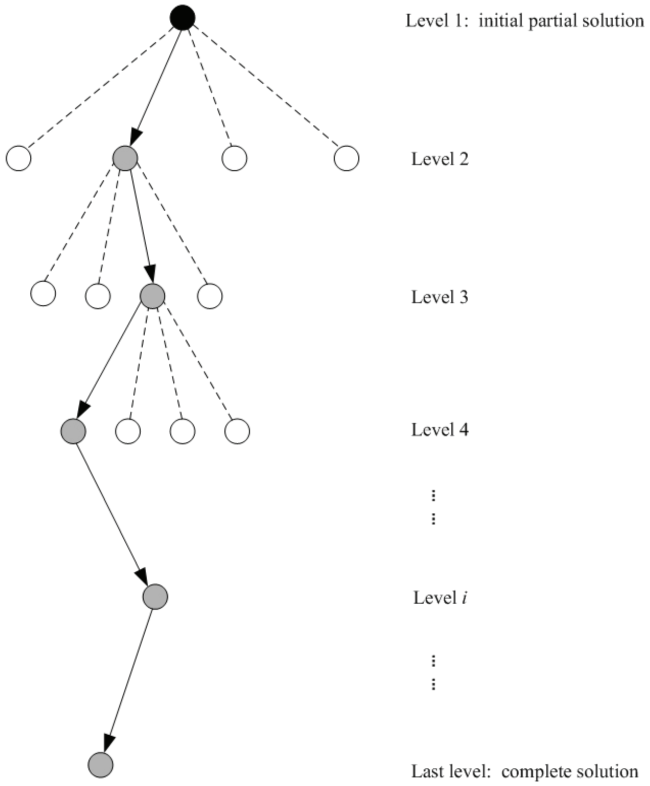
Level 1: initial partial solution
Level 2
Level 3
Level 4
Level i
Last level: complete solution

Fig. 1. Representation of basic greedy algorithm

Two important features of greedy method make it so popular are simple implementation and efficiency. Simple as it is, BG algorithm is highly efficient and sometimes it can produce an optimal solution for some optimization problem. For example, for problems such as activity-selection problem, fractional knapsack problem and minimum spanning trees problem, BG algorithm can obtain optimal solution by making a series of greedy choice. For these problems that the BG algorithm can obtain optimal solution, there is something in common: the optimal solution to the problem contains within it optimal solutions to sub-problems.
However, for other optimization problems that do not exhibit such property, the BG algorithm will not lead to optimal solution. Especially for the combinatorial optimization problems or NP-hard problem, the solution by BG algorithm is far away from satisfactory.

In BG algorithm, we make whatever choice seems best at the moment and then turn to solve the sub-problem arising after the choice is made. That is to say, the benefit is only locally evaluated. Consequently, even though we select the best at each step, we still missed the optimal solution. Just liking playing chess, a player who is focused entirely on immediate advantage is easy to be defeated, the player who can think several step ahead will win with more opportunity.
In this chapter, a novel greedy algorithm is introduced in detail, which is of some degree of forward-looking. In this algorithm, all the choices at the moment are evaluated more globally before the best one is selected. The greedy idea and enumeration strategy are both reflected in this algorithm, and we can adjust the enumeration degree so we can balance the efficiency and speed of algorithm.

2. Greedy Algorithm with forward-looking search strategy

To evaluate the benefit of a candidate choice more globally, an improved greedy algorithm with forward-looking search strategy (FG algorithm) was proposed by Huang *et al* [2], which was first proposed for tackling packing problem. It is a kind of growth algorithm and it is efficient for problem that can be divided into a series of sub-problems.
In FG algorithm, the promise of a candidate choice is evaluated not only by the current circumstance, but more globally by considering the quality of the complete solution that can be obtained from the partial solution represented by the node. The idea of FG algorithm can be illustrated by Fig.2:

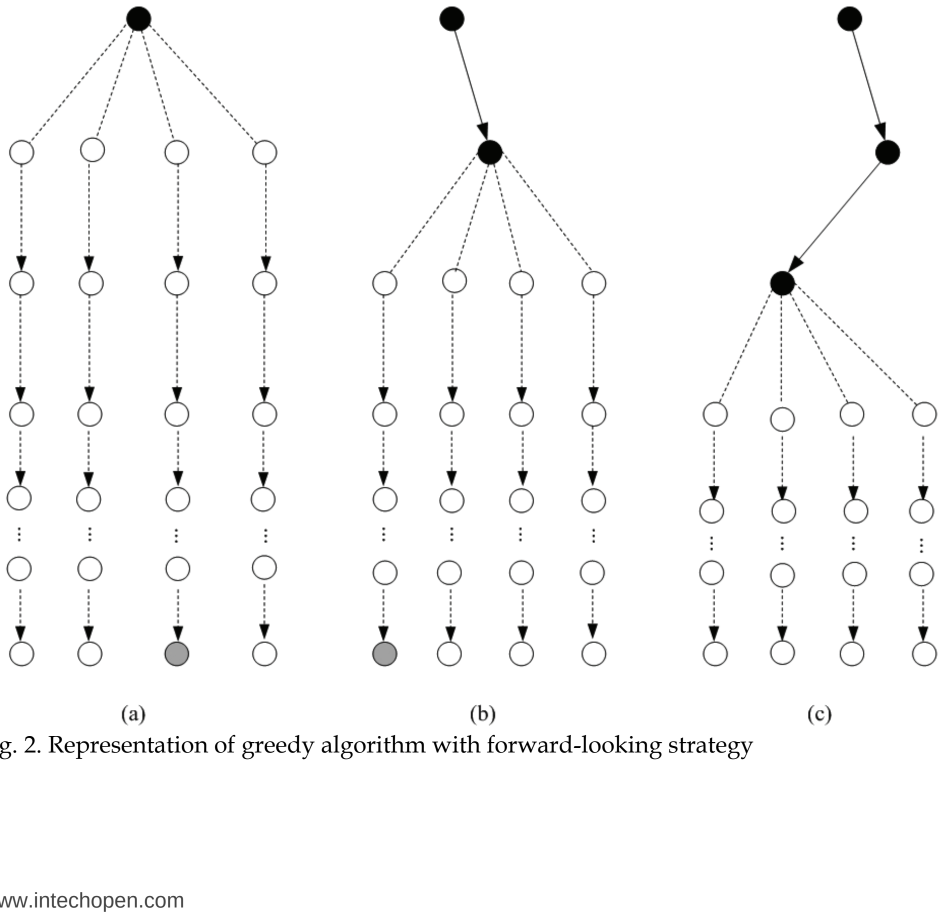

Fig. 2. Representation of greedy algorithm with forward-looking strategy

As shown in Fig.2 (a), there are four nodes at level 2 for the initial partial solution. We do not evaluate the promise of each node at once at the moment. Conversely, we tentatively update the initial partial solution by take the choices at level 2 respectively. For each node at level 2 (i.e., each partial solution at level 2), its benefit is evaluated by the quality of the complete solution resulted from it according to BG algorithm. From the complete solution with maximum quality, we backtrack it to the partial solution and definitely take this step. In other words, the node that corresponds to the complete solution with maximum quality (the gray circle in Fig.2 (a)) is selected as the partial solution. Then the search progresses to level 3. Level by level, this process is repeated until a complete solution is obtained.
After testing the global benefit of each node at current level, the one with great prospect will be selected. This idea can be referred as forward-looking, or backtracking. More formally, the procedure above can be described as follows:

```
Procedure FG (problem P)
Begin
    generate the initial partial solution S, and update P to a sub-problem;
    generate all current candidate choice as a list L;
    while (L is not empty AND finish condition is not met)
        max ⇐ 0
        for each choice c in L
            compute the global benefit: GloableBenefit (c, S, P);
            update max with the benefit;
        end for;
        modify S by selecting the choice that the global benefit equal to max;
        update P and L;
    end while;
End.
```

As shown in the above algorithm, in order to more globally evaluate the benefit of a choice and to overcome the limit of BG algorithm, we compute the benefit of a choice using BG itself in the procedure *GlobalBenefit* to obtain the so-called FG algorithm.
Similarly to BG algorithm, we start from the initial partial solution and repeat the above procedure until a complete solution is reached. Note that if there are several complete solutions with the same maximum benefit, we will select the first one to break the tie.
The global benefit of each candidate choice is described as:

```
Procedure GlobalBenefit (choice c, partial solution S, sub-problem P)
Begin
    let S′and P′ be copies of S and P;
    modify S′and P′ by taking the choice c;
    return BG(S, P);
End.
```

Given a copy *S*′ of the partial solution and a copy *P*′of sub-problem, then we update *S*′by taking the choice c. For the resulted partial solution and sub-problem, we use BG algorithm to obtain the quality of the complete solution.
It should be noted that Procedure FG only gives one initial partial solution. For some problems, there may be several choices for the initial partial solution. Similarly, the

Procedure *globalBenefit*() is implemented for the initial partial solutions respectively, and the one with maximum benefit should be selected.

3. Improved version of FG algorithm

3.1 Filtering mechanism

For some problems, the number of nodes is rather large at each level of search. Therefore, a filtering mechanism is proposed to reduce the computational burden. During filtering some nodes will not be given chance to be evaluated globally and be discarded permanently based on their local evaluation value. Only the remaining nodes are subject to global evaluation.

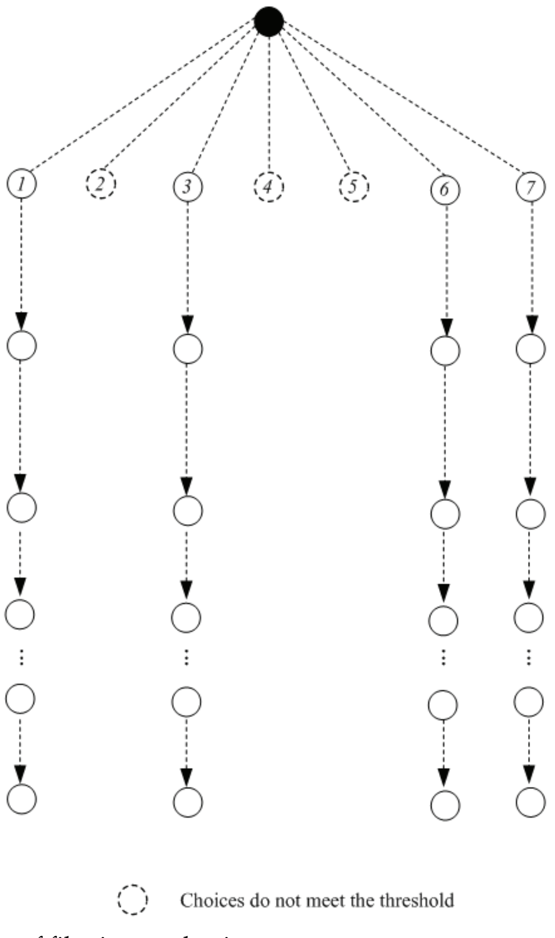

Fig. 3. Representation of filtering mechanism

As shown in Fig.3, there are 7 nodes at level 2. Firstly, the benefit of each node is locally evaluated. Then, only the promising nodes whose local benefit is larger than a given threshold parameter τ will be globally evaluated. The FG algorithm can be modified as FGFM algorithm:

```
Procedure FGFM (problem P)
Begin
    generate the initial partial solution S, update P to a sub-problem;
    generate all current candidate choice as a list L;
    while (L is not empty AND finish condition is not met)
        $max \Leftarrow 0$
        for each choice c in L
            if (local benefit > parameter $\tau$ )
                compute the global benefit: GloableBenefit (c, S, P);
                update max with global benefit;
            end if;
        end for;
        modify S by selecting the choice that the global benefit equal to max;
        update P and L;
    end while;
End.
```

Obviously, the threshold parameter τ is used to control the trade-off between the quality of the result and the computational time. If τ is set to be large enough, algorithm FGFM turns to be a BG algorithm; If τ is set to be small enough, algorithm FGFM turns to be a FG algorithm.

3.2 Multiple level enumerations

In the FG algorithm, the benefit of a node is globally evaluated by the quality of corresponding complete solution, which is resulted from the node level by level according to the BG algorithm. In order to further improve the quality of the solution, the forward-looking strategy can be applied to several levels.

This multi-level enumeration can be illustrated by Fig.4. For the initial partial solution, there are three candidate choices at level 2. From each node at level 2, there are several branches at level 3. Then we use procedure *GlobalBenefit* () to evaluate the global benefit of each nodes at level 3. That is to say, the three nodes at level 2 have several global benefits. We will choose the highest one as its global benefit. Afterwards, the one with the maximum global benefit from the three nodes at level 2 are selected as the partial solution.

If the number of enumeration levels is equal to (last level number - current level number-1) for each node, the search tree will become a complete enumeration tree, the corresponding solution of which will surely be optimal solution. However, the computational time complexity is unacceptable. Usually, the number of enumeration levels ranges from 1 to 4.

Obviously, the filtering mechanism and multi-level enumeration strategy are the means to control the trade-off between solution quality and runtime effort.

4. Applications

FG algorithm has been successfully applied to job shop scheduling problem [3], circle packing problem [2, 4] and rectangular packing problem [5]. In this section, the two-dimensional (2D) rectangle packing problem and its corresponding bounded enumeration algorithm is presented.

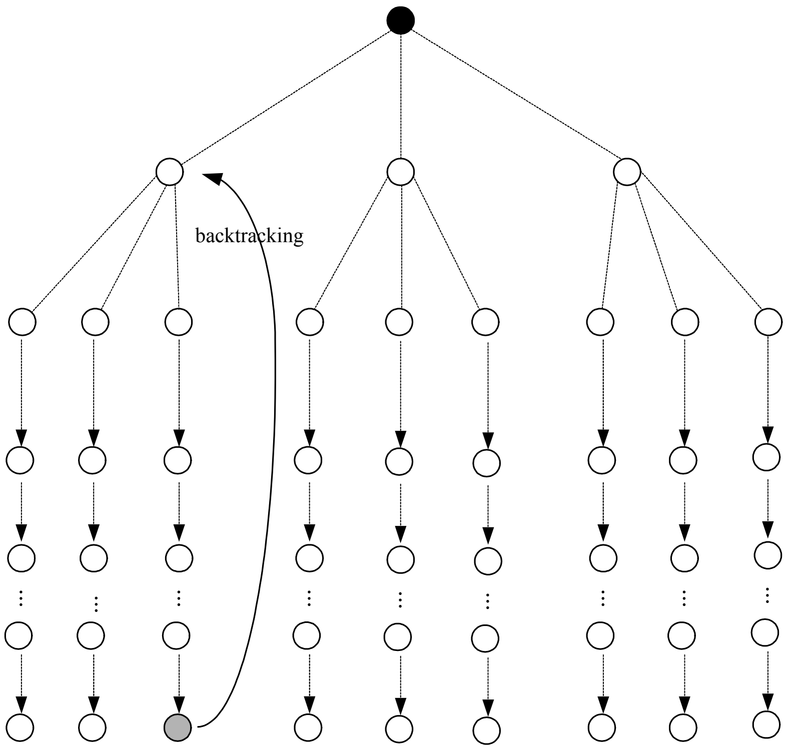

Fig. 4. The multi-level enumeration strategy

4.1 Problem definition

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.

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 *li* and width *wi*, $1 \le i \le n$, take the origin of the two-dimensional Cartesian coordinate system at the bottom-left corner of the container (see Fig.5). The aim of this problem is to determine if there exist a solution composed of *n* sets of quadruples $\{x_{11}, y_{11}, x_{12}, y_{12}\}$,…, $\{x_{n1}, y_{n1}, x_{n2}, y_{n2}\}$, 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 $1 \le i \le n$, 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 $1 \le i, j \le n, j \ne i$, rectangle i and j cannot overlap, i.e., one of the following condition should be met: $x_{i1} \ge x_{j2}$ or $x_j1 \ge x_i2$ or $y_{i1} \ge y_{j2}$ or $y_j1 \ge y_i2$;
3. $0 \le x_{i1}, x_i2 \le W$ and $0 \le y_{i1}, y_{i2} \le H$.

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 get rapidly 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.

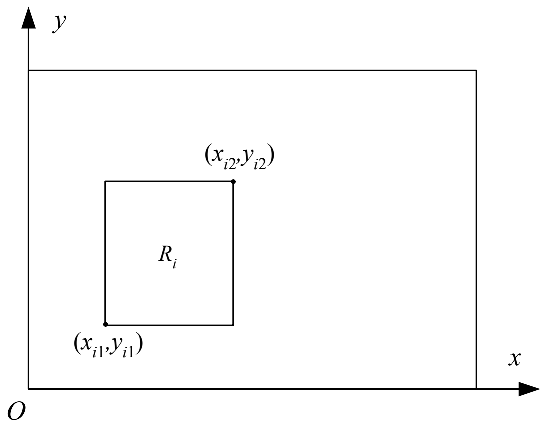

Fig. 5. Cartesian coordinate system

4.2 Preliminary

Definition *Configuration*. A configuration C is a pattern (layout) where m $(0 \le m < n)$ 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 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 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.

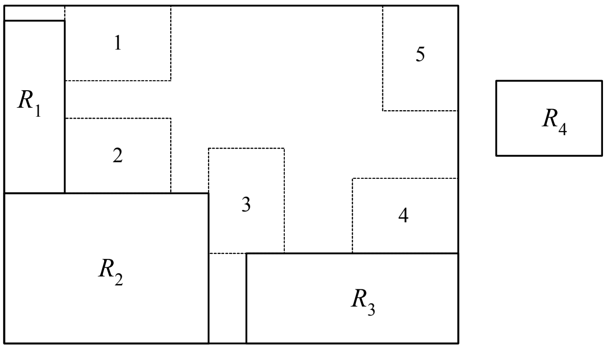

Fig. 6. Candidate corner-occupying action for rectangle R_4

Under current configuration, there may be several candidate packing positions for the current rectangle to be packed. At the configuration in Fig.6, three rectangles R_1, R_2 and R_3 are already placed in the container. There are totally 5 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.

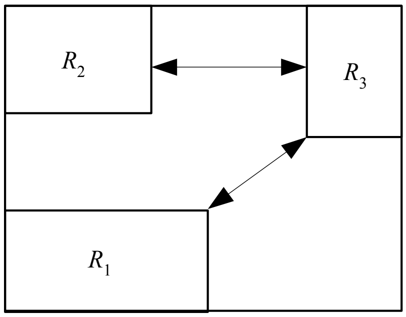

Fig. 7. Illustration of distance

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 arbitrary point on rectangle i and j, respectively. The minimal distance d_{ij} between i and j is:

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

In Fig.7, 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.7, 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:

$$\lambda = 1 - d_{\min}/(\frac{w_i + l_i}{2})$$

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,

$$d_{\min} = \min\{d_{ij} \mid j \in M \cup \{s_1, s_2, s_3, s_4\}, j \neq u, v\}$$

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 place 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. It is just the simple application of BG algorithm.

4.3 The basic algorithm: A_0

Based on the HDF rule and BG 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;
        end for;
        select the CCOA $(i, x, y, \theta)$ with the highest degree;
        modify $C$ by placing rectangle $i$ at $(x, y)$ with orientation $\theta$;
        modify $L$ according to the new configuration $C$;
    end while;
    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 orientationθ. 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.

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 doesn't 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.

4.4 The greedy algorithm with forward-looking strategy: 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 BenefitA_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 if-else
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$\theta$and call A_0 to reach a final configuration. If A_0 stops with success then *Benefit*A_1 returns a successful configuration,

otherwise *BenefitA*$_1$ 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, *BenefitA*$_1$ evaluates all existing CCOAs in L.
Now, using the procedure *BenefitA*$_1$, 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 ($i, x, y, \theta$) in $L$
            $d$= BenefitA$_1$ ($i, x, y, \theta, C, L$);
            if ($d$ is a successful configuration)
                stop with success;
            else
                update the maximum benefit with $d$;
            end if-else;
        end for;
        select the CCOA ($i^*, x^*, y^*, \theta^*$) with the maximum benefit;
        modify $C$ by placing rectangle $i^*$ at ($x^*, y^*$) with orientation $\theta^*$;
        modify $L$ according to the new configuration $C$;
    end while;
    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.

4.5 Computational complexity

We analysis the complexity of A_1 in the worst case, that is, when it cannot find 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 calculus 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 *BenefitA*$_1$ 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 *BenefitA*$_1$ though to the end each time. At every iteration of A_1, *BenefitA*$_1$ 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, *BenefitA*$_1$ does not continue to try other CCOAs, nor A_1 to exhaust the search space. In fact, every call to A_0 in *BenefitA*$_1$ 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 such 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.

4.6 Computational results

The set of tests is done using the Hopper and Turton instances [6]. 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. 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 best meta-heuristic (SA+BLF) in [6], HR [7], LFFT [8] and SPGAL [9]. 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.

As shown in Table 2, with the increasing of the number of rectangles, the running time of the two meta-heuristics and LFFT increases rather fast. HR is a fast algorithm, whose time complexity is only O(n3) [7]. Unfortunately, the running time of each instance for SPGAL is not reported in the literature. The mean time of all test instances for SPGAL is 139 seconds, which is acceptable in practical applications. It can be seen that A_1 is also a fast algorithm. Even for the problem instances of larger size, A_1 can yield solutions of high density within short running time.

It has shown from Table 2 that the running time of A_1 does not consistently accord with its theoretical time complexity. For example, the average time of C3 is 1.71 seconds, while the average time of C4 and C5 are both within 0.5 seconds. 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.

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 1. Computational results of our algorithm for the test instances from Hopper and Turton instances

Class	SA+BLF[1]		HR[2]		LFFT[3]		SPGAL[4]		A_1[5]	
	Gap	Time	Gap	Time	Gap	Time	Gap	Time (s)	Gap	Time
C1	4.0	42	8.33	0	0.0	1	1.7	–	0.00	0.37
C2	6.0	144	4.45	0	0.0	1	0.0	–	0.00	0.61
C3	5.0	240	6.67	0.03	1.0	2	2.2	–	1.07	1.71
C4	3.0	1980	2.22	0.14	2.0	15	0.0	–	1.64	0.15
C5	3.0	6900	1.85	0.69	1.0	31	0.0	–	1.09	0.40
C6	3.0	22920	2.5	2.21	1.0	92	0.3	–	0.83	3.98
C7	4.0	250800	1.8	36.07	1.0	2150	0.3	–	0.41	45.02
Average gap (%)	4.0		3.97		0.86		0.64		0.72	

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

[1] PC with a Pentium Pro 200MHz processor and 65MB memory [11].
[2] Dell GX260 with a 2.4 GHz CPU [15].
[3] PC with a Pentium 4 1.8GHz processor and 256 MB memory [14].
[4] The machine is 2GHz Pentium [16].
[5] 2.4 GHz PC with 512 MB memory.

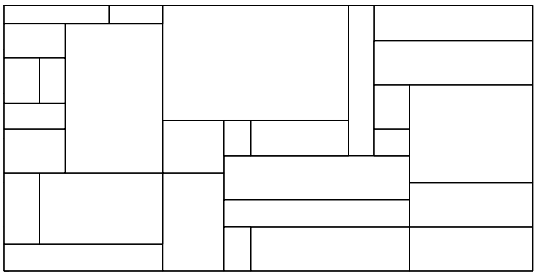
Fig. 8. Packing result of C31

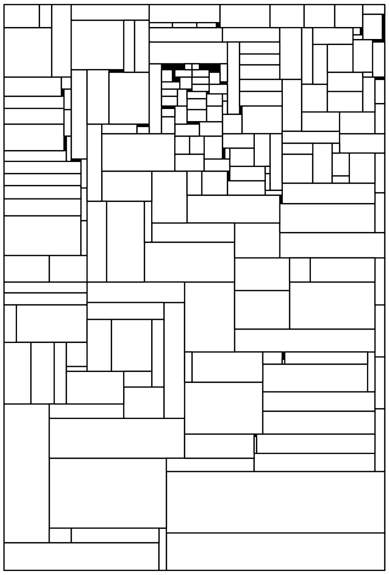
Fig. 9. Packing result of C73

In addition, we give the packing results on test instances C31 and C73 for A_1 in Fig.8~Fig.9. Here, the packing result of C31 is of optimal height, and the height C73 are only one length unit higher than the optimal height

5. Conclusion

The algorithm introduced in this chapter is a growth algorithm. Growth algorithm is a feasible approach for combinatorial optimization problems, which can be solved step by step. After one step is taken, the original problem becomes a sub-problem. In this way, the problem can be solved recursively. For the growth algorithm, the difficulty lies in that for a sub-problem, there are several candidate choices for current step. Then, how to select the most promising one is the core of growth algorithm.

By basic greedy algorithm, we use some concept to compute the fitness value of candidate choice, then, we select one with highest value. The value or fitness is described by quantified measure. The evaluation criterion can be local or global. In this chapter, a novel greedy

algorithm with forward-looking strategy is introduced, the core of which can more globally evaluate a partial solution.
For different problems, this algorithm can be modified accordingly. This chapter gave two new versions. One is of filtering mechanism, i.e., only part of the candidate choices with higher local benefit will be globally evaluated. A threshold parameter is set to allow the trade-off between solution quality and runtime effort to be controlled. The higher the threshold parameter, the faster the search will be finished., and the lower threshold parameter, the more high-quality solution may be expected. The other version of the greedy algorithm is multi-level enumerations, that is, a choice is more globally evaluated.
This greedy algorithm has been successfully used to solve rectangle packing problem, circle packing problem and job-shop problem. Similarly, it can also be applied to other optimization problems.

6. Reference

[1] A. Dechter, R. Dechter. On the greedy solution of ordering problems. ORSA Journal on Computing, 1989, 1: 181-189
[2] W.Q. Huang, Y Li, S Gerard, *et al*. A "learning from human" heuristic for solving unequal circle packing problem. Proceedings of the First International Workshop on Heuristics, Beijing, China, 2002, 39-45.
[3] Z. Huang, W.Q. Huang. A heuristic algorithm for job shop scheduling. Computer Engineering & Appliances (in Chinese), 2004, 26: 25-27
[4] W.Q. Huang, Y. Li, H. Akeb, *et al*. Greedy algorithms for packing unequal circles into a rectangular container. Journal of the Operational Research Society, 2005, 56: 539-548
[5] M. Chen, W.Q. Huang. A two-level search algorithm for 2D rectangular packing problem. Computers & Industrial Engineering, 2007, 53: 123-136
[6] E. Hopper, B.Turton, An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem. European J. Oper. Res, 128 (2001): 34-57
[7] D.F. Zhang, Y. Kang, A.S. Deng. A new heuristic recursive algorithm for the strip rectangular packing problem. Computers & Operational Research. 33 (2006): 2209-2217
[8] Y.L. Wu, C.K. Chan. On improved least flexibility first heuristics superior for packing and stock cutting problems. Proceedings for Stochastic Algorithms: Foundations and Applications, SAGA 2005, Moscow, 2005, 70-81
[9] A. Bortfeldt. A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces. European Journal of Operational Research. 172 (2006): 814-837