五一数学建模竞赛

承　诺　书

我们仔细阅读了五一数学建模竞赛的竞赛规则。

我们完全明白，在竞赛开始后参赛队员不能以任何方式（包括电话、电子邮件、网上咨询等）与本队以外的任何人（包括指导教师）研究、讨论与赛题有关的问题。

我们知道，抄袭别人的成果是违反竞赛规则的，如果引用别人的成果或其它公开的资料（包括网上查到的资料），必须按照规定的参考文献的表述方式在正文引用处和参考文献中明确列出。

我们郑重承诺，严格遵守竞赛规则，以保证竞赛的公正、公平性。如有违反竞赛规则的行为，我们愿意承担由此引起的一切后果。

我们授权五一数学建模竞赛组委会，可将我们的论文以任何形式进行公开展示（包括进行网上公示，在书籍、期刊和其他媒体进行正式或非正式发表等）。

参赛题号（从 A/B/C 中选择一项填写）：B

参赛队号：B19129436

参赛组别（研究生、本科、专科、高中）：本科

所属学校（学校全称）：武汉大学

参赛队员：　队员 1 姓名：刘文韬

队员 2 姓名：党政

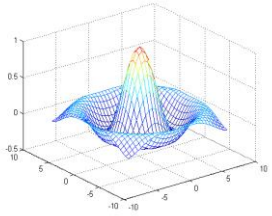
队员 3 姓名：王昊

联系方式：　Email：793082695@qq.com　联系电话：18876221931

日期：　2019 年 5 月 4 日

（除本页外不允许出现学校及个人信息）

五 一 数 学 建 模 竞 赛



题 目：基于剪枝-遗传算法的木板切割优化模型

摘要：

“徐州家具厂”问题，本质上是二维优化下料问题的一种，而优化下料问题模型的建立，从计算的复杂性上来讲，这是一个 NP(Nondeterministic Polynomial) 难题,难以精确求解。而智能优化算法（如遗传算法，模拟退火法，群集智能算法等），往往可以极大的缩短问题求解时间。因此，我们在遗传算法的基础上提出了一种剪枝-遗传算法（PGA，Pruning Genetic Algorithm），结合遗传算法的高鲁棒性、易扩展性和剪枝算法的高性能，并以此为基础，使用改进的最低水平线搜索排样算法，应用多维整数规划，给出不同情境约束下的问题解决方案。

对于问题一、问题二，属于原始的优化下料问题，目标是空间利用率（SU，Space Utilization）最高。我们对每个产品的水平、竖直摆放分别进行基因编码，并依据先后排列顺序组成染色体，作为 PGA 算法的初始种群，使用 SU 作为个体的适应度值（FV，Fitness Value），然后使用改进的最低水平线搜索排样算法对每个个体计算 FV，最后通过 FV 导向的多代交叉、变异、选择的过程，获得最优解或较优解。对于问题一，我们计算出的 SU 为 98.30%；对于问题二，我们得到的 SU 从高到低依次为 98.30%、97.03%、96.63%。

对于问题三、问题四，属于需求约束的优化下料问题，目标是在满足生产数量要求的前提下，使 SU 达到最高，我们的做法是使用 PGA 算法和改进的最低水平线搜索排样算法求出候选的单块木板较优解，然后使用 MATLAB 进行多维整数规划，获得最优解；其中对于多类型产品问题，额外补充 PGA 算法的低维较优解，进一步优化“边角料”的利用率。对于问题三，我们使用 4 种切割方案，得到 92.86%的总利用率；对于问题四，我们使用 5 种切割方案，得到 93.07%的总利用率。

问题五是以利润最大为目标的优化下料问题，PGA 算法利用遗传算法的易扩展性，巧妙地将总利润作为 FV，求得单块木板利润最大的最优解。又由于本问题无生产任务的约束，所以单块木板的最优解即可用于所有木板，共获得 117410 元的总利润，同时总利用率为 98.30%。

最后，我们对模型的优缺点进行了分析，并给出了一些改进措施。

关键词：改进的最低水平线搜索排样法、剪枝-遗传算法（PGA）、多维整数规划、“边角料”优化

一、问题重述

徐州某家具厂日前新进一批木板，其规格如表 1 所示。对于后续的家具加工的工程需要，该厂需要将这批新进木板进行分割，满足共计四类产品的生产任务，四类产品的产品尺寸及生产任务如表 2 所示。

表 1 木板的尺寸

木板	长度(mm)	宽度(mm)
S1	3000	1500

表 2 产品尺寸及生产任务

产品名称	长度(mm)	宽度(mm)	生产任务(件)	利润(元/件)
P1	373	201	774	19.9
P2	477	282	2153	23.0
P3	406	229	1623	21.0
P4	311	225	1614	16.0

现要求保证对木板的切割方案为最优切割方案这一前提下，建立数学模型，依次解决以下问题。

问题 1：建立数学模型，在一块木板上只切割 P1 这一类产品，得到木板利用率最高（也即剩余木板面积最小）的最优方案，并确定产品数和利用率。

问题 2：在一块木板上切割 P1 和 P3 这两类产品，依次得到所有切割方案中木板利用率前三高的方案及相应参数。

问题 3：在木板上切割 P1 和 P3 这两类产品，并要求在保证 P1 和 P3 的生产任务能够完成的情况下，得到在木板总利用率最高的情况下各木板的切割方案及利用率。

问题 4：在木板上切割 P1-P4 四类产品，并要求在保证各产品的生产任务能够完成的情况下，得到木板总利用率最高的情况下各木板的切割方案及利用率。

问题 5：在不考虑产品的生产任务和利用率的情况下，用 100 块木板切割 P1-P4 四类产品，得到木板总利润最大的情况下各木板的切割方案及相关参数。

二、问题分析

“徐州家具厂”问题，从本质上，是二维优化下料问题的变形，而优化下料问题模型的建立, 从计算的复杂性上来讲, 这是一个 NP(Nondeterministic Polynomial) 难题,难以精确求解。而智能优化算法[1]（如遗传算法[4]，模拟退火法，群集智能算法等），往往可以极大的缩短问题求解时间。因此，我们提出了一种剪枝-遗传算法（PGA，Pruning Genetic Algorithm），并以此为基础，使用改进的最低水平线搜索排样算法，应用多维整数规划，给出不同情境约束下的问题解决方案。

问题一、问题二属于原始的优化下料问题，对于产品的排列，优化目标为空间

利用率（SU，Space Utilization）最高。我们对每个产品的水平、竖直摆放分别进行基因编码，并依据先后排列顺序组成染色体，作为 PGA 算法的初始种群，然后使用改进的最低水平线搜索排样算法计算每个个体的适应度值（FV，Fitness Value），随机筛选优秀个体交配、变异至稳定后，获得最优解（注：以下最优解均指本模型最优解）或较优解。

问题三、问题四属于需求约束的优化下料问题，对于产品的排列，是在满足生产数量要求的前提下，使 SU 尽量达到最高，我们的做法是使用 PGA 算法+改进的最低水平线搜索排样算法求出候选的单块木板较优解，然后使用多维整数规划规划每种木板数量，求出最优解；同时对于多类型产品问题，补充 PGA 算法的低维较优解，优化“边角料”的利用率。

问题五是以利润最大为目标的优化下料问题，将 PGA 算法中的适应度函数改为利润，便可用 PGA 算法求得单块木板利润最大的最优解。同时由于无生产任务的约束，单块木板的最优解即可用于所有木板。

三、问题假设

1、假设木板的厚度和切割时割缝的宽度均可忽略不计。
2、假设产品以正交且平行于木板边缘的方式排放。
3、假设木板无缺陷，均可作为产品底料。
4、假设产品生产合格率为 100%，且不会在制作过程中损坏。

四、符号说明

变换符号	符号说明
$p_1, p_2 \cdots p_i \cdots p_N$	被排列产品的编号序列
l_i	产品 p_i 的长
w_i	产品 p_i 的宽
L	木板长度
W	木板宽度
η_{SU}	单块木板利用率
a_m	编号为 m 的产品种类排样量
π_m	编号为 m 的产品种类单位利润
Π	单块木板总利润
h_k	第 k 个最高水平线的高度

H_j	j 个最高水平线的高度所构成的集合
w_{h_k}	第 k 个最高水平线的宽度
W_{h_j}	j 个最高水平线的宽度所构成的集合
G	由染色体组成的基因种群集合
g_i	基因种群中第 i 个染色体
p_{ij}	基因种群中第 i 个染色体的第 j 个基因
F_v	染色体的适应度

五、模型的建立

5.1 单块木板切割问题

对于单块木板切割成多个产品的优化下料问题，可以将之转化为：寻找某一确定的产品序列，在给定的排样方法下将之依次在木板上排放，尽量保持相邻产品之间的间隙最小，最终使得排放结果能够尽可能地布满整个木板。

设共有 n 种产品种类，因为待排产品存在着横放和竖放两种方法，不妨设第 k 种产品横放对应着编号 $2k-1$，竖放对应着编号 $2k$。则可以对一固定的产品序列进行编码，编号为 $p_1, p_2 \cdots p_i \cdots p_N$（$p_i \in [1,2n] \cap Z, 1 \le i \le N$）。其中，第 i 个产品 p_i 的长为 l_i（$1 \le i \le N$）宽为 w_i（$1 \le i \le N$），该序列产品在长为 L 宽为 W 的木板上依次进行排样。

在排样过程中，以木板左下角为坐标原点建立坐标系，并设第 i 个排样产品（$1 \le i \le N$），它的左下角坐标为 (x_i, y_i)，则它们所占用的区域为 S_i，其中

$$S_i = \{(x,y) | \ x_i < x < x_i + l_i, y_i < y < y_i + w_i\}$$

则对于排入第 $i+1$ 个产品，需满足

$$\begin{cases} \bigcap_{m=1}^{i+1} S_m = \varnothing \\ x_{i+1} + l_{i+1} < L, y_{i+1} + w_{i+1} < W \end{cases}$$

在利用率最大化问题中，我们定义利用率 η_{SU} 为所有已排列的矩形的总面积与底板总面积之比，即

$$\eta_{SU} = \frac{\sum_{i=1}^{N} l_i w_i}{LW}$$

并使得 η_{SU} 尽可能达到最大。

在利润最大化问题中，设最终木板上编号为 m（$1 \le m \le n$）实际排样 a_m 块，其每类板子对应的单位利润为 π_m 每件。则定义总利润 Π 为各类板子存在数与单位利润的乘积的和，即为

$$\Pi=\sum_{m=1}^{N} a_m \pi_m$$

所以在不同的最优目标下，对木板各种最优的切割方案总是可以概括为给定排样方法下不同的排样序列。所以我们将各类问题转化为在不同的目标或限制条件下，寻找某一排样序列，其所排列出的解为最优解。

5.2 搜索排样算法

5.2.1 最低水平线搜索排样算法

对于给定的产品排列序列 $p_1, p_2 \cdots p_i \cdots p_N$（$1 \le i \le N$），在保证木板利用率达到最大的前提下，我们采取最低水平线搜索排样算法，即运用对最低水平线不断更替的方法来进行排列[2]。

其基本步骤如下：

(1) 对于已排列的 i 个产品所构成的排样，将其高轮廓线选出。

(2) 于其中自最低的水平线开始依次进行考察，检测第 $i+1$ 个产品的宽度是否小于该水平线的宽度，且放置后是否新的排样会超出木板的轮廓。

(3) 当产品可以被放入则于此处放入，对下一产品重复步骤(2)的考察。反之则基于下一水平线完成步骤(2)的考察，直到出现一个水平线能够将该产品放入。

(4) 若各水平线上均不能放入该产品，则考察产品序列中下一产品，再重复步骤(2)和(3)的考察，直到产品序列中各元素被考察完毕。

5.2.2 改进的最低水平线搜索排样算法

基于欲解决问题的实际情况，我们对模型进行优化改进。其基本步骤如下：

(1) 假设目前已由 i 个产品（$1 \le i \le i+1 \le N$）根据最低水平线法构成了一组排样，那么显然，该排样结构的最高轮廓线可以拆分为 j 个最高水平线（$1 \le j \le i+1$），它们的高度构成集合 $H_j = \{h_k \mid k = 1, 2 \cdots j\}$。集合中各元素所对应的宽度 w_{h_k}（$1 \le k \le j$）构成了一个新的集合 $W_{h_j} = \{w_{h_k} \mid k = 1, 2 \cdots j\}$。

我们运用优先级队列（priority queue）对集合 W_{h_j} 中各元素进行排样，将宽度 w_{h_k}（$1 \le k \le j$）所对应的水平线 h_k（$1 \le k \le j$）较低的排定义为优先级较高，然后按照优先级从高到低依次排序生成一个优先级队列 W_j，显然，该队列各宽度元素的前后顺序与其所对应的水平线高度的低高排列相匹配。因产品插入前后集合 W_{h_j} 的变化较小，而显然队列 W_j 中最靠前的元素即排样各最高水平线集合中最低水平线的宽度，则其不仅便于对集合 W_{h_j} 中的元素查找，且利于集合 H_j 和 W_{h_j} 的动态维护。

(2) 在下一块产品 p_{i+1} 放置前，设 $k=1$。则考察以下两个条件，

$$\begin{cases} w_{i+1} + h_k \le W \\ l_{i+1} \le w_{h_k} \end{cases}, k = 1, 2 \cdots n$$

若均满足则可排放该矩形，将之排放。

(3) 若该条件式组不满足，因该问题中被排列产品的种类较少，所以我们将 p_{i+1} 所对应产品的其他产品种类的尺寸，并依次检验是否满足此条件式组。

若有至少一种产品满足该条件式组，则选出其中最适合被放入的产品，即 $|w_{h_k} - w_{i+1}|$ 最小，并将之替换为新的 p_{i+1}；这样做的话可以使得被排列矩形间的缝隙

更小，利于我们更快地找到我们所想要的序列。

(4) 若各种类均不能满足此条件式组时，则对优先级队列W_j中下一个元素进行考察，即令$k=k+1$并重复步骤(2)和(3)的考察，依此类推，直至将该产品排入原排样中为止。

(5) 若对k的各个取值都不能满足产品p_{i+1}的插入时，则对这一序列进行剪枝处理，记将$p_1, p_2 \cdots p_i$直接作为序列。（见 5.3.4）

在求解问题 1 的过程中，我们分别对未改进以及改进后的最低水平线搜索排样算法进行了运用，得到的结果如图 1，具有较大的性能优化

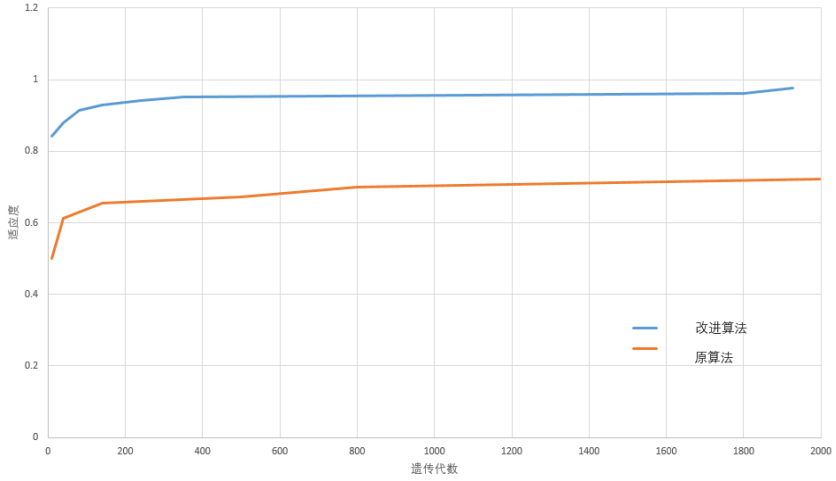


图 1 改进算法与原算法的对比图

5.3 剪枝-遗传算法（PGA）

5.3.1 算法流程

采用 PGA 算法求解木板切割问题的算法流程如图 2 所示。

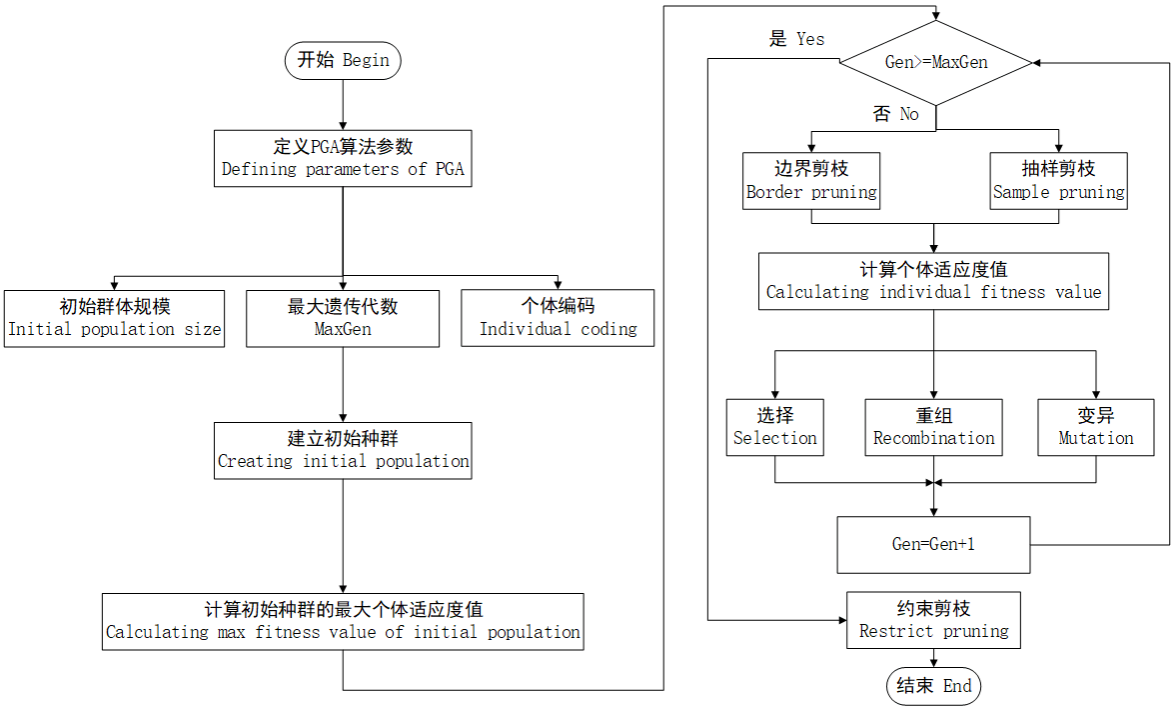


图 2 PGA 算法流程图

5.3.2 基因编码策略

在排样方法给定后，每一组产品序列则可以编码为一个染色体，产品序列中对每一位上产品的编码可以视为一个基因。根据十进制策略[3]，对一组产品序列中每一个待排产品进行整数编排，并在编排中考虑到待排产品的横放与竖放问题。

我们默认序列内每种编码对应的矩形只能横放，则不妨设第 k 种产品横放对应着编号 $2k-1$，竖放对应着编号 $2k$。如 $2,5,\cdots,p_N$ 序列代表的是 $\{2,5,\cdots,p_N\}$ 染色体，即先竖直排放 1 号产品，随后水平排放 3 号产品，依此类推。

设共有 n 种产品类型，则基因可表示序列 $p_1,p_2\cdots p_i\cdots p_N$（$p_i\in[1,2n]\cap Z,1\le i\le N$）。此时每个染色体均为一种产品序列，即对单个木板的一种产品排列方案，也即一种木板切割的下料方案。

5.3.3 初始化种群

设共有 M 个染色体，则各个染色体可依某种顺序定义为 $g_1,g_2\cdots g_i\cdots g_M$（$1\le i\le M$），全部染色体构成了一个集合，称为基因种群 G。对于集合 G 中各染色体元素的基因产生采用随机生成的办法，设第 i 个染色体 g_i 为 $\{p_{i1},p_{i2}\cdots p_{ij}\cdots p_{iN}\}$（$1\le i\le M,1\le j\le N$），其中 p_{ij} 为第 i 个染色体的第 j 个基因。则通过函数

$$p_{ij}=[rand()*2n]$$

对染色体中各基因进行随机生成。

5.3.4 适应度函数

适应度函数是用于评估各染色体优劣的评估函数，进而区分优劣。对于本问题中，因各个染色体均对应着一种对于木板的切割方案，所以其适应度函数则显然由最优目标所给出，如在求解最高利用率问题中，其适应度函数即为木板总利用率，即

$$F_v=\eta_{SU}=\frac{\sum_{i=1}^{N}l_iw_i}{LW}$$

显然有 $F_v\in[0,1]$，且 F_v 越高越理想。

同样，在求解最高利润问题时，其适应度函数即为木板总利润，即

$$F_v=\Pi=\sum_{m=1}^{N}a_m\pi_m$$

此时有 $F_v\in[0,+\infty)$，且 F_v 越高越理想，算法实现如图 3 所示（全部代码见附录 4）：

```
#ifdef PROFIT
    for (member = 0; member < GROUP_SCALE; member++)
    {
        int sum = Cutting(Population[member].Xn);
        double sum_pro = 0;
        for (int i = 0; i < sum; i++) {
            sum_pro += Profit[(size_t)Population[member].Xn[i]];
        }
        Population[member].Fitness = sum_pro;
    }

#else
    for (member = 0; member < GROUP_SCALE; member++)
    {
        int sum = Cutting(Population[member].Xn);
        double sum_area = 0;
        for (int i = 0; i < sum; i++) {
            sum_area += Genetics[(size_t)Population[member].Xn[i]].Area;
        }
        Population[member].Fitness = sum_area / B_AREA;
    }

#endif // PROFIT
```

图 3 适应度函数代码

总之，将对适应度函数结果的高低能够与最终判定染色体对应方案的优劣时刻呈成正比关系即可。

5.3.5 基因传递

当初始化染色体种群后，设根据第i次基因传递后各染色体所构成的集合为G_i。每一代基因传递过程中，先通过交叉算子，随机决定该代染色体集合G_i中染色体的交叉生成情况，进而组成新的染色体集合，而后通过变异算子随机改变集合G_i中的部分元素，最后通过选择算子在集合中选择，得到第$i+1$代基因传递后的染色体集合G_{i+1}。

5.3.5.1 轮盘赌算法

轮盘赌算法是我们用于随机确定染色体的方法，其原理是染色体适应度越高被选择的概率越大。设染色体集合G中共含有M个染色体。根据第i个染色体（$1\le i\le M$）的适应度$F_v(g_i)$，因染色体对应着一种排板方案，而显然我们更希望越优良的方案被选择进行基因传递的概率越大，或者其与另一优良方案结合生成更优方案。则可设第i个染色体被选中的概率为

$$P(g_i)=\frac{F_v(g_i)}{\sum_{i=1}^{M}F_v(g_i)}$$

显然可以得到

$$\begin{cases}P(g_i)\in[0,1]\\ \sum_{i=1}^{M}F_v(g_i)=1\end{cases}$$

随机在 0~1 中生成一个有理数r，令$k=1,2,\cdots M$并依次判断以下条件

$$\sum_{i=1}^{k-1}F_v(g_i)<r<\sum_{i=1}^{k}F_v(g_i)$$

满足条件的k对应的染色体则被选择。

5.3.5.2 交叉算子

通过轮盘赌算法来确定进行交叉的染色体，将其不放回的选择出，直至构成一个含有适量偶数个数元素的待交叉染色体集合，并实行单点交叉算子。即对群体中的个体进行两两随机配对，每对中随机决定交叉点位置，而后两染色体互换交叉点位置后的全部基因。

5.3.5.3 变异算子

依旧按照轮盘赌算法来确定进行变异的染色体，并随机对染色体中的某一基因进行修改。因为我们编码的时候将产品的横竖放考虑到其中，同时可以同时实现两张变异算子：旋转变异和位置变异[5]。即对基因的修改既可能存在着对排样产品的种类发生改变，也可能存在着对排样的方向发生改变。

5.3.5.4 选择算子

因染色体的交叉和变异是随机的，所以可能会破坏基因优良的染色体，即丢失较优的产品排样序列。所以在选择出下一代集合时，为保证上代集合中较优的产品排列序列能大概率被，我们运用随机联赛选择法，即随机在 $0\sim M$ 中随机生成一个自然数r，而后在染色体集合G_i中选取适用度最高的前r个染色体。将其有放回的选择出并重复这一操作，直到构成一个新的染色体集合G_{i+1}。

同样，为了保证最优的排列序列可能被丢失的情况，我们在选择算子结束后，将新一代G_{i+1}中适应度最低的染色体替换为上一代G_i中适应度最高的染色体，这样既保证最优解不会被丢失，也可以便于我们快速定位找到最优解。

5.3.6 剪枝优化

剪枝优化是在算法运行的过程中，自动的对一些不合理的方案提前去除，避免在迭代过程中继续扩大较差解或者一些不必要的计算。虽然剪枝算法小概率存在破坏全局最优解，但是带来的时间优化增益是非常明显的，同时在较大规模问题上，依然可以达到非常逼近全局最优的结果，所以经常被用来优化 NP 问题。

我们针对优化下料算法，实现了以下几种剪枝方式：

5.3.6.1 边界剪枝

在排列产品的过程中，由于最低水平线的迭代会将新产生的水平线压入队列底部，即使该水平线处于木板边界，明显无法继续放置产品，还是会对可能产生的结果进行判断、搜索，这是可以规避的。

设产品种类有n种，已排样结构的最高水平线构成了高度集合H_j和宽度集合W_{h_j}。则可得各种类产品的长度最小值$l_{\min}$和宽度最小值$w_{\min}$，各水平线中高度最低值$h_{\min}$和宽度最小值$w_{h\min}$，即有

$$\begin{cases} l_{\min}=\min\{l_1,l_2\cdots l_{2n}\} \\ w_{\min}=\min\{w_1,w_2\cdots w_{2n}\} \\ h_{\min}=\min\{h_1,h_2\cdots h_j\} \\ w_{h\min}=\min\{w_{h_1},w_{h_2}\cdots w_{h_j}\} \end{cases}$$

我们在水平线出队列时，即完成了第i个产品的放置时，先对其进行一次剪枝判断，判断条件如下：

$$\begin{cases} l_{\min}+h_{\min}>W \\ w_{\min}>w_{h\min} \end{cases}$$

若满足条件，则删除该染色体的后续部分，即保留序列的前 i 项。

5.3.6.2 抽样剪枝

在优化排放算法搜索可放置产品时，如果遍历整个产品库，在产品数目较大的时会带来不必要的开销，同时由于产品库最符合产品不变，可能会导致结果早熟，所以我们为了解决这个问题，提出了基于随机的搜索剪枝，每次从产品库随机取 N 个产品，然后在 N 个产品中选择最适合的产品排放，以此增加基因的多样性。

5.3.6.3 约束剪枝

在特定约束下的优化下料问题，某些较优解虽然空间利用率较高，但是会对约束条件造成冲击，例如需求{P1：500 件、P2：100 件}，得到的解为单个木板{P1：2 件、P2：40 件}，在规划过程中是无法用到的，所以针对类似的情况，我们提出了约束剪枝，将非常不平衡的染色体删除。

若产品种类有 n 种，则设 q_i 为第 i 类产品的需求量，c_i 为该类产品在某一染色体对应方案的下料所得量。则可比较 $q_1,q_2\cdots q_i\cdots q_n$ 和 $c_1,c_2\cdots c_i\cdots c_n$ 两组序列的相关性。设相对系数序列为 $\alpha_1,\alpha_2\cdots\alpha_i\cdots\alpha_n$，其中 α_i 为两组序列第 i 类产品的相对系数，即

$$\alpha_i=\frac{c_i}{q_i},q_i>0,i=1,2\cdots n$$

可求得两组序列的相关系数 α。

$$\alpha=\sum_{i=1}^{n}(\alpha_i-\frac{\sum_{i=1}^{n}\alpha_i}{n})$$

设相关系数的阈值为 $\alpha_{\max}$，则剪枝的判断判断条件为

$$\alpha>\alpha_{\max}$$

若满足则去除该种方案。

5.4 有需求约束的优化下料问题

与无需求约束的优化下料问题相比，有需求约束的优化下料问题增加了产品生产任务需求量这一约束条件，优化目标变为使用的木板数量最少。

在无需求约束的优化下料问题中，PGA 算法能够有效求解，并生成包括最佳方案在内的多种切割方案，但每个方案中的各产品生产数量无法人为控制。因此，无法直接使用 PGA 算法求解有需求约束的最少木板切割问题。

我们取单块木板切割的前 N（$N>2$）种较优解，并以各个方案的木板数量作为变量，可将该问题转化为一个线性规划问题。

对于多类型产品问题，仅仅依靠线性规划和单板切割是不够的，所有产品均包括而又满足需求比例的单板切割往往利用率较差，所以我们使用 PGA 算法算出从单产品，双产品，三产品，一直到多产品的较优解，在线性规划时进行局部优化，提高“边角料”的利用率。

线性规划的优化目标为木板数量，即每种方案的木板数量之和。线性规划方程如下：

$$f(x)=\sum_{i=1}^{m}x_i$$

$$\text{s.t.}\begin{cases}\sum_{i=1}^{m} x_i \cdot a_{ij} \geq task_j \\ x_i \geq 0\end{cases}$$

其中 $f(x)$ 是线性规划的目标函数，即满足需求的木板用量，m 是切割方案总数，x_i 是采用第 i 种切割方案的木板 S1 数量，a_{ij} 是第 i 种切割方案中生产的第 j 种产品数量，$task_j$ 为第 j 种产品生产任务数量（$j \in \{1,2,3,4\}$）。约束条件为每种产品的总数大于等于生产任务的需求，且 x_i 为非负整数。

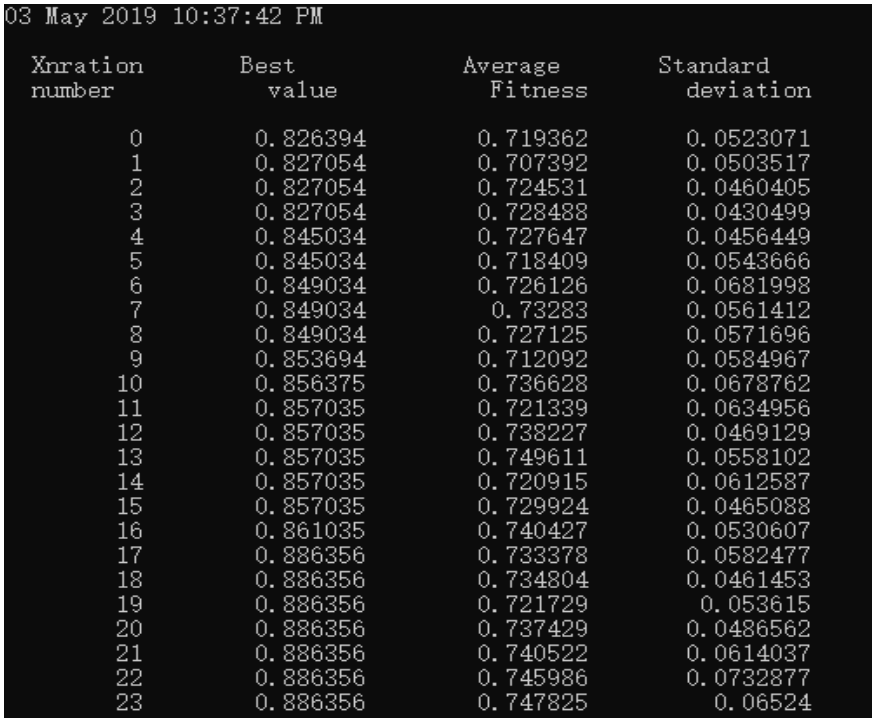
六、问题求解

6.1 问题 1 的求解

在问题 1 中，我们只需要在 S1 大小的木板上排列 P1 这一种大小的产品，因为产品的长与宽不一致，我们将产品的横放与竖放两种情况定义成两种产品，分别设为 1 和 2。而这作为排放序列 $p_1, p_2 \cdots p_i \cdots p_N$ 的元素取值，即：

$$p_i \in [1,2] \cap Z, 1 \leq i \leq N$$

采用 PGA 算法，调整相应算法参数（种群数量 50，最大遗传代数 4000），程序运行时会输出各代的最优适应度，平均适应度以及适应度标准差以供参考，运行时的一个界面如图 4：



03 May 2019 10:37:42 PM

Xnration number	Best value	Average Fitness	Standard deviation
0	0.826394	0.719362	0.0523071
1	0.827054	0.707392	0.0503517
2	0.827054	0.724531	0.0460405
3	0.827054	0.728488	0.0430499
4	0.845034	0.727647	0.0456449
5	0.845034	0.718409	0.0543666
6	0.849034	0.726126	0.0681998
7	0.849034	0.73283	0.0561412
8	0.849034	0.727125	0.0571696
9	0.853694	0.712092	0.0584967
10	0.856375	0.736628	0.0678762
11	0.857035	0.721339	0.0634956
12	0.857035	0.738227	0.0469129
13	0.857035	0.749611	0.0558102
14	0.857035	0.720915	0.0612587
15	0.857035	0.729924	0.0465088
16	0.861035	0.740427	0.0530607
17	0.886356	0.733378	0.0582477
18	0.886356	0.734804	0.0461453
19	0.886356	0.721729	0.053615
20	0.886356	0.737429	0.0486562
21	0.886356	0.740522	0.0614037
22	0.886356	0.745986	0.0732877
23	0.886356	0.747825	0.06524

图 4 PGA 算法运行图

得到的最优结果如表 3：

表 3 问题 1 的结果

P1 的数量	木板利用率
59	98.30%

对最优切割方案可视化，其切割方案如图 5(具体数值见附录 1，下同)：

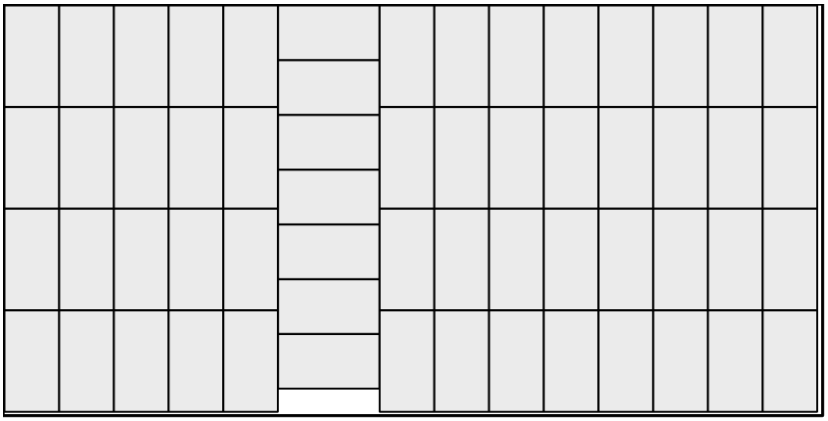


图 5 问题一最佳切割方案

从上图可以看出，木板面积得到充分利用，仅有极少数不可利用的余料。

6.2 问题 2 的求解

在问题 2 中，我们需要在 S1 大小的底板上排列 P1 和 P3 这两种大小的矩形，同样，我们根据矩形可以横放和竖放两种情况，将可放置的矩形类型定义为四种矩形，分别设为 1、2、3、4，即有：

$$p_i \in [1,4] \cap Z, 1 \le i \le N$$

采用 PGA 算法求解，得到木板利用率最高的 3 种切割方案如下：

表 4 问题 2 的结果

方案编号	P1 的数量	P3 的数量	木板利用率
1	59	0	98.2979%
2	57	1	97.0319%
3	58	0	96.6319%

其切割方案分别如下：

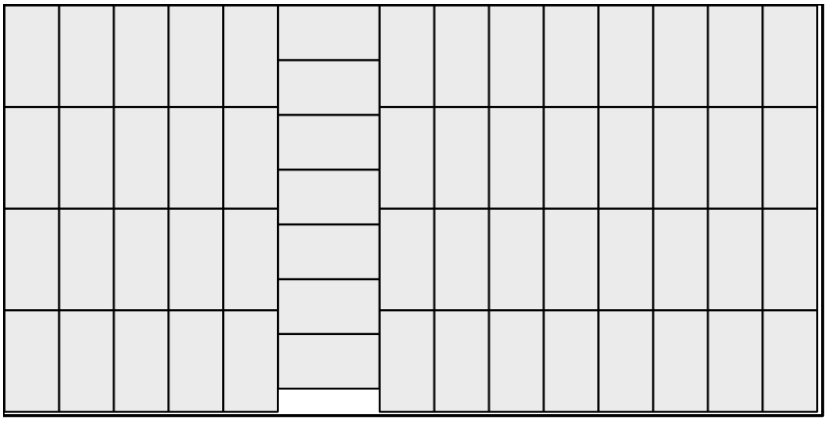


图 6 问题二切割方案一

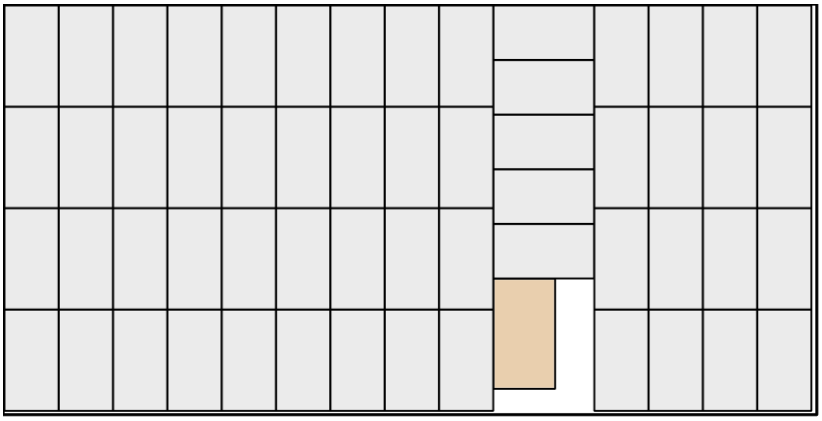


图 7 问题二切割方案二

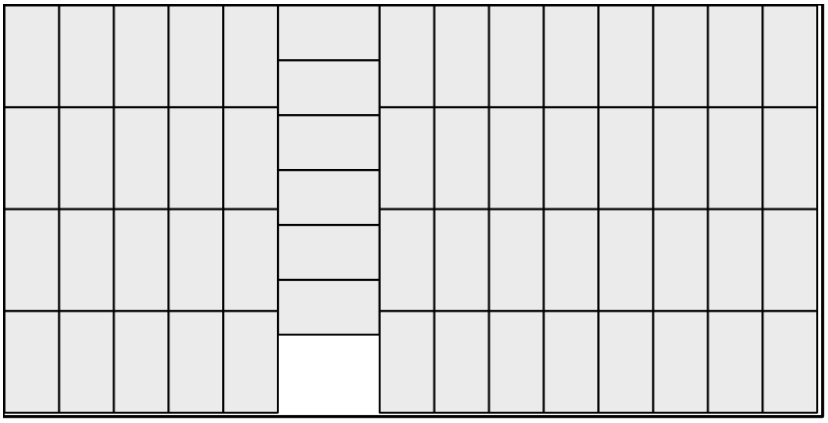


图 8 问题二切割方案三

从图中可以直观地看出木板利用率前 3 的三种切割方案的不同和优劣。

6.3 问题 3 求解

问题 3 要求满足 P1、P3 产品的生产任务的木板总利用率最高的切割方案，由于生产任务数量一定，只要使用的木板总数最少，木板总利用率最高。求解该问题的线性规划方程如下所示：

$$\min f(x)=\sum_{i=1}^{m} x_i$$

$$\text{s.t.}\begin{cases}\sum_{i=1}^{m} x_i \cdot a_{i1} \geq 774 \\ \sum_{i=1}^{m} x_i \cdot a_{i3} \geq 1623 \\ x_i \geq 0\end{cases}$$

其中利用 PGA 算法生成的单块木板切割方案（只包含 P1、P3）共有 47 种，$m=47$；a_{i1} 为第 i 种单块木板切割方案中 P1 产品数量，a_{i3} 为第 i 种单块木板切割方

案中 P3 产品数量，具体见附录 2。使用 MATLAB 中的整数线性规划函数 *intlinprog* 进行求解，结果如表 5 所示：

表 5 问题 3 的结果

木板 S1 数量	P1 数量	P3 数量	木板利用率	备注
34	3	42	91.77%	每块木板切割方案相同
11	59	0	98.30%	同上
4	0	44	90.91%	同上
1	23	19	77.58%	同上
合计数量：50	774	1623	木板总利用率:92.86%	木板总利用率=所有产品的总面积/所有木板的总面积

6.4 问题 4 求解

问题 4 要求满足 P1、P2、P3、P4 产品的生产任务的木板总利用率最高的切割方案。求解该问题的线性规划方程如下所示：

$$\min f(x)=\sum_{i=1}^{m} x_i$$

$$\text{s.t.}\begin{cases}\sum_{i=1}^{m} x_i \cdot a_{i1} \geq 774\\ \sum_{i=1}^{m} x_i \cdot a_{i2} \geq 2153\\ \sum_{i=1}^{m} x_i \cdot a_{i3} \geq 1623\\ \sum_{i=1}^{m} x_i \cdot a_{i4} \geq 1614\\ x_i \geq 0\end{cases}$$

其中利用 PGA 算法生成的单块木板切割方案共有 67 种，$m=67$；a_{ij} 为第 i 种单块木板切割方案中第 j 种产品的数量，具体见附录 2。使用 MATLAB 中的整数线性规划函数 *intlinprog* 进行求解，结果如表 6 所示：

表 6 问题 4 的结果

木板 S1 的数量	P1 的数量	P2 的数量	P3 的数量	P4 的数量	木板 利用率	备注
2	0	0	0	59	91.75%	每块木板切割方案相同
57	0	0	28	24	95.17%	同上
75	5	28	0	0	92.03%	同上
8	48	6	0	0	97.91%	同上
2	7	0	0	52	92.52%	同上
1	1	5	0	0	16.61%	切割方案同第 3 种，只取部分
1	0	0	27	24	93.10%	切割方案同第 2 种，只取部分
合计数量：146	774	2153	1623	1614	木板 总利用率: 93.07%	木板总利用率 =所有产品的总面积/所有木板的总面积

6.5 问题 5 的求解

在问题 5 中，空间利用率和生产任务不做要求，所以我们只需保证单个木板上排列的利润最大，将遗传算法中的适用度函数

$$F_v = \eta_{SU} = \frac{\sum_{i=1}^{N} l_i w_i}{LW}$$

改为

$$F_v = \Pi = \sum_{m=1}^{N} a_m \pi_m$$

最后我们得到如下结果：

表 7 问题 5 的结果

木板 S1 的数量	P1 的数量	P2 的数量	P3 的数量	P4 的数量	利润	木板 利用率	备注
100	59	0	0	0	117410	98.30%	每块木板切割方案相同
木板 S1 合计数量 100	5900	0	0	0	总利润: 117410	木板 总利用率: 98.30%	木板总利用率 =所有产品的总面积/所有木板的总面积

七、模型的评价与改进

7.1 模型评价

优点：

(1) 算法效率高。我们对最低水平线搜索排样算法和遗传算法都进行了改进和优化，显著降低了算法时间复杂度，提高了效率。

(2) 通用性强。本模型可求解优化下料问题中常见的无需求约束、有需求约束最少用料问题和最大利润问题。只需修改相应的原料尺寸数据、约束条件即可用于求解其他同类问题，具有较强的通用性和实用性。

缺点：

(1) 考虑的实际问题有所简化、抽象。模型仅考虑了在原料上正交且平行于边缘的切割方案，在实际生活中可能存在的非正交或不平行于边缘的问题无法用本模型求解。

(2) 问题 3、问题 4 求解采用线性规划，尽管包括了 PGA 算法生成的绝大部分较优的单块木板分割方案，但仍可能遗漏组成全局最优解的方案，无法保证求得的结果最优，也无法验证结果是否最优。

(3) 当使用的木板总数相同时，采用线性规划求解的结果无法保证切割方式最少，在实际生活中无法保证生产成本最低。

7.2 模型改进

(1) 对最低水平线搜索排样算法进一步改进，使其支持非正交的排样方式，从而能够求解切割方式非正交或不平行于边缘的下料问题。

(2) 使用 GPU 并行计算加速求解。尽管 PGA 算法已对遗传算法进行了优化，但受 CPU 性能及线程数量限制，种群数量和遗传代数。采用 GPU 并行计算可极大加速算法求解，可使用更大的种群数量和遗传代数，从而极大减少遗漏较优解的可能性。

八、参考文献

[1] Sweeney P.E. Cutting and Packing problem : a Categorized application-Orientated research Bibliography.Journal of the operational research,1998,88-96

[2] Liu H，Zhou J，Wu X，et al，Optimization algorithm for rectangle packing problem based on varied-factor genetic algorithm and lowest front-line strategy，2014 IEEE Congress on Evolutionary Computation (CEC):2014

[3]李明，黄平捷，周泽魁．基于小生境遗传算法的矩形件优化排样．湖南大学学报（自然科学版），2009(1)：46-49．

[4] Zhao X F，Cui Y D，Yang Y，et al，A genetic algorithm for rectangle strip packing problem， Journal of Computer-Aided Design ＆ Computer Graphics:2008 20 (4): 540 － 544．

[5] Hopper E , Turton B.Genetic algorithm for a 2D industrial packing problem .Computers and Industrial Engineering,1999, 37(1):375 -378 .

附录

附 1：问题 1 中产品 P1 的排列方式（由下文代码直接生成）

第 1 好的方案为：适应度 0.982979
第 0 个零件 P1 ： 左下角坐标为(0,0),放置角度是水平.
第 1 个零件 P1 ： 左下角坐标为(201,0),放置角度是水平.
第 2 个零件 P1 ： 左下角坐标为(402,0),放置角度是水平.
第 3 个零件 P1 ： 左下角坐标为(603,0),放置角度是水平.
第 4 个零件 P1 ： 左下角坐标为(804,0),放置角度是水平.
第 5 个零件 P1 ： 左下角坐标为(1005,0),放置角度是水平.
第 6 个零件 P1 ： 左下角坐标为(1206,0),放置角度是水平.
第 7 个零件 P1 ： 左下角坐标为(1407,0),放置角度是水平.
第 8 个零件 P1 ： 左下角坐标为(1608,0),放置角度是水平.
第 9 个零件 P1 ： 左下角坐标为(1809,0),放置角度是水平.
第 10 个零件 P1 ： 左下角坐标为(2010,0),放置角度是竖直.
第 11 个零件 P1 ： 左下角坐标为(2383,0),放置角度是水平.
第 12 个零件 P1 ： 左下角坐标为(2584,0),放置角度是水平.
第 13 个零件 P1 ： 左下角坐标为(2785,0),放置角度是水平.
第 14 个零件 P1 ： 左下角坐标为(2010,201),放置角度是竖直.
第 15 个零件 P1 ： 左下角坐标为(2383,373),放置角度是水平.
第 16 个零件 P1 ： 左下角坐标为(2584,373),放置角度是水平.
第 17 个零件 P1 ： 左下角坐标为(2785,373),放置角度是水平.
第 18 个零件 P1 ： 左下角坐标为(1005,373),放置角度是水平.
第 19 个零件 P1 ： 左下角坐标为(1809,373),放置角度是水平.
第 20 个零件 P1 ： 左下角坐标为(1608,373),放置角度是水平.
第 21 个零件 P1 ： 左下角坐标为(1407,373),放置角度是水平.
第 22 个零件 P1 ： 左下角坐标为(1206,373),放置角度是水平.
第 23 个零件 P1 ： 左下角坐标为(804,373),放置角度是水平.
第 24 个零件 P1 ： 左下角坐标为(0,373),放置角度是水平.
第 25 个零件 P1 ： 左下角坐标为(603,373),放置角度是水平.
第 26 个零件 P1 ： 左下角坐标为(402,373),放置角度是水平.
第 27 个零件 P1 ： 左下角坐标为(201,373),放置角度是水平.
第 28 个零件 P1 ： 左下角坐标为(2010,402),放置角度是竖直.
第 29 个零件 P1 ： 左下角坐标为(2010,603),放置角度是竖直.
第 30 个零件 P1 ： 左下角坐标为(1005,746),放置角度是水平.
第 31 个零件 P1 ： 左下角坐标为(1608,746),放置角度是水平.
第 32 个零件 P1 ： 左下角坐标为(2383,746),放置角度是水平.
第 33 个零件 P1 ： 左下角坐标为(2785,746),放置角度是水平.
第 34 个零件 P1 ： 左下角坐标为(1809,746),放置角度是水平.
第 35 个零件 P1 ： 左下角坐标为(2584,746),放置角度是水平.
第 36 个零件 P1 ： 左下角坐标为(1407,746),放置角度是水平.
第 37 个零件 P1 ： 左下角坐标为(804,746),放置角度是水平.
第 38 个零件 P1 ： 左下角坐标为(603,746),放置角度是水平.

第 39 个零件 P1 ： 左下角坐标为(1206,746),放置角度是水平.
第 40 个零件 P1 ： 左下角坐标为(402,746),放置角度是水平.
第 41 个零件 P1 ： 左下角坐标为(201,746),放置角度是水平.
第 42 个零件 P1 ： 左下角坐标为(0,746),放置角度是水平.
第 43 个零件 P1 ： 左下角坐标为(2010,804),放置角度是竖直.
第 44 个零件 P1 ： 左下角坐标为(2010,1005),放置角度是竖直.
第 45 个零件 P1 ： 左下角坐标为(2785,1119),放置角度是水平.
第 46 个零件 P1 ： 左下角坐标为(2584,1119),放置角度是水平.
第 47 个零件 P1 ： 左下角坐标为(1005,1119),放置角度是水平.
第 48 个零件 P1 ： 左下角坐标为(2383,1119),放置角度是水平.
第 49 个零件 P1 ： 左下角坐标为(1809,1119),放置角度是水平.
第 50 个零件 P1 ： 左下角坐标为(1608,1119),放置角度是水平.
第 51 个零件 P1 ： 左下角坐标为(1407,1119),放置角度是水平.
第 52 个零件 P1 ： 左下角坐标为(603,1119),放置角度是水平.
第 53 个零件 P1 ： 左下角坐标为(402,1119),放置角度是水平.
第 54 个零件 P1 ： 左下角坐标为(804,1119),放置角度是水平.
第 55 个零件 P1 ： 左下角坐标为(201,1119),放置角度是水平.
第 56 个零件 P1 ： 左下角坐标为(0,1119),放置角度是水平.
第 57 个零件 P1 ： 左下角坐标为(1206,1119),放置角度是水平.
第 58 个零件 P1 ： 左下角坐标为(2010,1206),放置角度是竖直.
P1 零件共使用 59 个
总共使用零件 59 个。

附 2：线性规划候选切割方案

只包含 P1、P3 的单块木板切割方案

P1	P3
0	44
1	43
3	42
4	41
5	40
5	39
6	38
6	37
8	36
10	35
11	34
12	33
12	32
14	31
15	30
17	29
18	28

包含 P1、P2、P3、P4 的单块木板切割方案

P1	P2	P3	P4
19	16	0	0
0	18	0	15
23	0	16	0
33	0	0	9
0	11	20	0
0	0	14	27
59	0	0	0
0	0	44	0
0	30	0	0
0	0	0	59
0	0	28	24
0	0	22	29
0	0	19	33
0	0	26	24
0	0	23	28
0	0	20	32
0	0	21	31

19	27
21	26
24	25
25	24
27	23
28	22
30	21
31	20
33	19
33	18
35	17
36	16
36	15
38	14
40	13
41	12
41	11
43	10
44	9
44	8
46	7
48	6
49	5
49	5
51	4
54	2
55	1
55	2
57	1
59	0
52	3

0	0	25	26
0	0	22	30
0	0	23	29
0	0	24	28
1	30	0	0
3	29	0	0
5	28	0	0
6	27	0	0
7	26	0	0
9	25	0	0
10	24	0	0
12	23	0	0
14	22	0	0
15	21	0	0
18	20	0	0
20	19	0	0
20	18	0	0
22	17	0	0
24	16	0	0
28	15	0	0
49	2	0	0
42	6	0	0
52	2	0	0
47	5	0	0
51	3	0	0
53	2	0	0
50	4	0	0
52	3	0	0
48	6	0	0
46	0	0	11
1	0	0	58
2	0	0	56
3	0	0	55
4	0	0	54
6	0	0	53
7	0	0	52
8	0	0	49
8	0	0	48
11	0	0	47
12	0	0	45
14	0	0	44
16	0	0	42
18	0	0	40

20	0	0	38
21	0	0	36
47	0	0	10
49	0	0	8
51	0	0	6
55	0	0	2
56	0	0	1

附3：线性规划代码（Matlab）

```
num = 67
a = csvread('data.csv', 1)
A = a' * -1
f = ones(num, 1)
b = [-774; -2153; -1623; -1614]
lb = zeros(num, 1)
[x, fval, exitflag] = intlinprog(f, [1:num], A, b, [], [], lb)


for i=1:num
    if x(i) > 0
        disp([num2str(i), ' ', num2str(x(i))])
    end
end
calc(774, 2153, 1623, 1614, fval)
fval
```

附 4：PGA 代码（C++）

```
//main.cpp
#include <iostream>
#include "Pruning_Genetic.h"

extern Individual Population[GROUP_SCALE + 1];
#ifdef HighThree
extern Individual High[];
#endif // HighThree
extern ofstream out;

int main()
{
```

```
13.     int Xnration;
14.     int i;
15.     int seed = 142;
16.
17.     showTime();
18.     initGroup(seed);
19.     evaluate();
20.     selectBest();
21.
22.     for (Xnration = 0; Xnration < MAX_GENS; Xnration++)
23.     {
24.         selector(seed);
25.         crossover(seed);
26.         mutate(seed);
27.         report(Xnration);
28.         evaluate();
29.         elitist();
30.     }
31.
32.     cout << "\n";
33.     cout << "  Best member after " << MAX_GENS << " Xnrations:\n";
34.     cout << "\n";
35.
36.     for (i = 0; i < N_VARS; i++)
37.     {
38.         cout << "  X(" << i + 1 << ") = " << Population[GROUP_SCALE].Xn[i] << "\
    n";
39.     }
40.     cout << "\n";
41.     cout << "  Best Fitness = " << Population[GROUP_SCALE].Fitness << "\n";
42.
43. #ifdef HighThree
44.     for (int j = TIME - 1; j > -1; j--) {
45.         cout << "第" << TIME - j << "好的方案为：适应度
    " << High[j].Fitness << endl;
46.     }
47.     for (int j = TIME - 1; j > -1; j--) {
48.         out << "第" << TIME - j << "好的方案为：适应度
    " << High[j].Fitness << endl;
49.         showCutting(High[j].Xn);
50.     }
51. #else
52.     showCutting(Population->Xn);
53. #endif //HighThree
```

```
54.
55.     showTime();
56.     while (1);
57.     return 0;
58. }
59. //Pruning_Genetic.h
60. //pure genetic algorithm is from:https://blog.csdn.net/Touch_Dream/article/details/68066055
61.
62.
63. #ifndef _GENETIC_H_
64. #define _GENETIC_H_
65. #define HighThree
66. # include <fstream>
67. using namespace std;
68. //#define PROFIT
69.
70. #ifdef HighThree
71. #define TIME 20
72. #endif // HighThree
73.
74.
75. #define  PI    3.14159265358979323846
76.
77. //遗传算法参数，种群规模（0~100）、繁殖代数、函数变量个数、交叉概率、变异概率
78. # define GROUP_SCALE    50
79. # define MAX_GENS       5000
80. # define N_VARS         80
81. # define P_MATING       0.8
82. # define P_MUTATION     0.20
83.
84.
85. //木板参数：木板长度，宽度，面积，四种零件的长度宽度，零件种类数,最小零件尺寸，使用的零件数
86. #define B_H     1500
87. #define B_W     3000
88. #define B_AREA  4500000
89. #define P1_L    373
90. #define P1_W    201
91. #define P2_L    406
92. #define P2_W    229
93. #define P3_L    477
94. #define P3_W    282
95. #define P4_L    311
```

```
#define P4_W    225
#define P_Num   8
#define P_MIN_WIDTH 201
#define TRY_TIME 4

//基因编码
struct GCode {
    double Area;//面积
    double Length;//长度
    double Width;//宽度
};

//每种零件有竖直，水平两种摆放
extern struct GCode Genetics[P_Num];


extern double Profit[P_Num];


struct Individual
{
    int Xn[N_VARS];      //存放变量值
    double Fitness;          //适应值
    double ReFitness;        //适应值概率密度
    double SumFitness;       //累加分布，为轮盘转

};
struct X_Range
{
    int Upper;           //变量的上界取值
    int Lower;           //变量的下界取值
};

template<typename T>
T randT(T Lower, T Upper); //产生任意类型随机数函数

void crossover(int& seed);
void elitist();        //基因保留
void evaluate();

void initGroup(int& seed);

void selectBest();
void mutate(int& seed);
```

```
140.
141. double r8_uniform_ab(double a, double b, int& seed);
142. int i4_uniform_ab(int a, int b, int& seed);
143.
144. void report(int Xnration);
145. void selector(int& seed);
146. void showTime();
147. void Xover(int one, int two, int& seed);
148. int Cutting(int* Xn);
149. int showCutting(int* Xn);
150. inline int CanBePut(int Lwidth, int Plength, int Pwidth, int Lheight);
151.
152. void randperm(int N, int* a);
153.
154. #endif //
155. //Pruning_Genetic.cpp
156. #define _CRT_SECURE_NO_WARNINGS
157.
158. # include <cstdlib>
159. # include <iostream>
160. # include <iomanip>
161.
162. # include <iomanip>
163. # include <cmath>
164. # include <ctime>
165. # include <cstring>
166. #include "Pruning_Genetic.h"
167. #include <queue>
168.
169.
170. //申请种群内存，其中多加 1 个是放置上一代中最优秀个体
171. struct Individual Population[GROUP_SCALE + 1];
172. #ifdef HighThree
173. Individual High[TIME];
174. #endif // HighThree
175.
176.
177.
178. ofstream out = ofstream("out.txt");
179. #ifdef PROFIT
180. double Profit[P_Num] = {
181.     23.0,23.0,19.9,19.9,21,21,16,16
182. };
183.
```

```
#endif // PROFIT

int P_order[P_Num/2] = { 1,3,4,2 };

struct GCode Genetics[P_Num] = {
										{74973,P1_L,P1_W},{74973,P1_W,P1_L},//1
										{92974,P2_L,P2_W},{92974,P2_W,P2_L},//3
										{69975,P4_L,P4_W},{69975,P4_W,P4_L},//4
										{134514,P3_L,P3_W},{134514,P3_W,P3_L}//2


};

X_Range  XnRange[N_VARS] = {
								{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},
								{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},
								{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},
								{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},
								{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},
								{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},
								{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},
								{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME},{0,TRY_TIME}

};

struct HLine
{
	int Width;//可用的宽度在变化
	int Base;//左下角横坐标
	int Height;//高度

};

//越低的越靠上
bool operator<(HLine a, HLine b) {
	if (a.Height > b.Height) return true;
```

```
220.     return false;
221. }
222. 
223. void swapReference(int* a, int i, int j) {
224.     int tmp = a[i];
225.     a[i] = a[j];
226.     a[j] = tmp;
227. }
228. 
229. void randperm(int N,int*a) {
230. 
231.     for (int i = 0; i < N; i++)
232.         a[i] = i;
233.     for (int i = 1; i < N; i++)
234.         swapReference(a, i, randT(0, i));
235. 
236. }
237. 
238. 
239. 
240. 
241. 
242. //打印排列结果
243. int showCutting(int* Xn) {
244.     vector<int> sum;
245.     for (int j = 0; j < P_Num; j++) {
246.         sum.push_back(0);
247.     }
248. 
249. 
250. 
251.     //std::priority_queue 实现的大根堆
252.     priority_queue<HLine> products;
253.     products.push(HLine{ B_W,0,0 });
254. 
255.     int i = 0;
256.     do {
257. 
258.         HLine lowline = products.top();
259.         products.pop();
260.         if (B_H - lowline.Height < P_MIN_WIDTH || lowline.Width < P_MIN_WIDTH)
    {//边界剪枝
261.             continue;
262.         }
```

```
263.
264.         // 1 宽度装 0 不能装
265.         if (CanBePut(lowline.Width, Genetics[(int)Xn[i]].Length, Genetics[(int
    )Xn[i]].Width, lowline.Height) == 1) {
266.             //可以放下那么
267.             //1.水平线宽度改变
268.             //2.加入新水平线
269.             int newwidth, newheight,oldbase;
270.
271.                 Genetics[(int)Xn[i]].Length;
272.                 newheight = Genetics[(int)Xn[i]].Length;
273.                 newwidth = Genetics[(int)Xn[i]].Width;
274.                 lowline.Width -= newwidth;
275.                 oldbase = lowline.Base;
276.                 lowline.Base += newwidth;
277.
278.             products.push(lowline);
279.             products.push(HLine{ newwidth,oldbase, newheight + lowline.Height
    });
280.             sum[Xn[i]]++;
281.             out << "第" << i << "个零件 P"<< P_order[Xn[i] / 2 ] << " : " << "左
    下角坐标为(" << oldbase << "," << lowline.Height << "),放置角度是
    " << ((Xn[i] + 1)%2 == 0 ? "竖直." : "水平.") << endl;
282.             i++;
283.
284.         }
285.         else {
286.
287.
288.             //不可以放下那么
289.             //1.对其他零件进行判断，取最接近的，可以就改变
290.             //2.都不可以就换水平线(继续循环)
291.             int best_width = lowline.Width;
292.             int best_Pid = -1;
293.
294.             //int a[TRY_TIME];
295.             //memset(a, 0, TRY_TIME * sizeof(int));
296.             //randperm(TRY_TIME, a);
297.
298.             for (int j = 0; j < TRY_TIME; j++) {//此处当零件过多时，改为随机算法，
    随机取 N 个
299.                 int k = j;
300.
301.
```

```
302.                 if (CanBePut(lowline.Width, Genetics[k].Length, Genetics[k].Wi
    dth, lowline.Height) == 1) {
303.                     int local_width = lowline.Width - Genetics[k].Width;
304.                     if (local_width < best_width) {
305.                         best_width = local_width;
306.                         best_Pid = k;
307.                     }
308.                 }
309.             }
310.
311.             if (best_Pid == -1) {//没有可以替换的
312.                 continue;
313.             }
314.             else {//有可以替换的
315.                 Xn[i] = best_Pid;
316.
317.                 int newwidth, newheight;
318.
319.                 Genetics[(int)Xn[i]].Length;
320.                 newheight = Genetics[(int)Xn[i]].Length;
321.                 newwidth = Genetics[(int)Xn[i]].Width;
322.                 lowline.Width -= newwidth;
323.                 lowline.Base += newwidth;
324.
325.                 products.push(lowline);
326.                 products.push(HLine{ newwidth,lowline.Base - newwidth, newheig
    ht + lowline.Height });
327.                 sum[Xn[i]]++;
328.                 out << "第" << i << "个零件
    P" << P_order[Xn[i]/2 ] << " : " << "左下角坐标为
    (" << lowline.Base - newwidth << "," << lowline.Height << "),放置角度是
    " << ((Xn[i] + 1) % 2 == 0 ? "竖直." : "水平.") << endl;
329.                 i++;
330.             }
331.         }
332.     } while (!products.empty());
333.     int Psum = 0;
334.     for (int j = 0; j < P_Num/2; j+=2) {
335.         out << "P" << P_order[j/2] << "零件共使用" << sum[j]+sum[j+1] << "个
    " << endl;
336.         Psum += sum[j]+sum[j+1];
337.     }
338.     out << "总共使用零件" << Psum << "个。"<<endl;
339.
```

```
    return i;
}


//改进的最低水平线算法计算对于 Xn[]能放多少零件，返回能放的零件个数。
int Cutting(int* Xn) {
    //std::priority_queue 实现的大根堆
    priority_queue<HLine> products;
    products.push(HLine{ B_W,0,0 });

    int i = 0;
    do {

        HLine lowline = products.top();
        products.pop();
        if (B_H - lowline.Height < P_MIN_WIDTH || lowline.Width < P_MIN_WIDTH) {//剪枝
            continue;
        }

        // 1 宽度装 0 不能装
        if (CanBePut(lowline.Width, Genetics[(int)Xn[i]].Length, Genetics[(int)Xn[i]].Width, lowline.Height) == 1) {
            //可以放下那么
            //1.水平线宽度改变
            //2.加入新水平线
            int newwidth, newheight;


                Genetics[(int)Xn[i]].Length;
                newheight = Genetics[(int)Xn[i]].Length;
                newwidth = Genetics[(int)Xn[i]].Width;
                lowline.Width -= newwidth;
                lowline.Base += newwidth;

            products.push(lowline);
            products.push(HLine{ newwidth,lowline.Base-newwidth, newheight + lowline.Height });
            i++;

        }
        else {
            continue;
```

```
381.             //不可以放下那么
382.             //1.对其他零件进行判断，取最接近的，可以就改变
383.             //2.都不可以就换水平线(继续循环)
384.             int best_width = lowline.Width;
385.             int best_Pid = -1;
386.
387.
388.             //int a[TRY_TIME];
389.             //memset(a, 0, TRY_TIME * sizeof(int));
390.             //randperm(TRY_TIME, a);
391.
392.             for (int j = 0; j < TRY_TIME; j++) {//此处当零件过多时，改为随机算法，
    随机取 N 个
393.                 int k = j;
394.
395.
396.                 if (CanBePut(lowline.Width, Genetics[k].Length, Genetics[k].Wi
    dth, lowline.Height) == 1) {
397.                     int local_width = lowline.Width - Genetics[k].Width;
398.                     if (local_width < best_width) {
399.                         best_width = local_width;
400.                         best_Pid = k;
401.                     }
402.                 }
403.             }
404.
405.             if (best_Pid == -1) {//没有可以替换的
406.                 continue;
407.             }
408.             else {//有可以替换的
409.                 Xn[i] = best_Pid;
410.
411.                 int newwidth, newheight;
412.
413.                     Genetics[(int)Xn[i]].Length;
414.                     newheight = Genetics[(int)Xn[i]].Length;
415.                     newwidth = Genetics[(int)Xn[i]].Width;
416.                     lowline.Width -= newwidth;
417.                     lowline.Base += newwidth;
418.
419.                 products.push(lowline);
420.                 products.push(HLine{ newwidth,lowline.Base - newwidth, newheig
    ht + lowline.Height });
421.                 i++;
```

```
422.             }
423.
424.
425.         }
426.     } while (!products.empty());
427.     return i;
428. }
429.
430. /*
431. inline int CanBePut(int Lwidth,int Plength,int Pwidth,int Lheight) {
432.     if (Lwidth >= Plength && (Lheight + Pwidth) <= B_H) //能装下长度
433.         return 2;
434.     else //不能装下长度
435.         if (Lwidth >= Pwidth && (Lheight + Plength) <= B_H) //能装下宽度
436.             return 1;
437.         else
438.             return 0;
439. }
440. */
441.
442. inline int CanBePut(int Lwidth, int Plength, int Pwidth, int Lheight) {
443.         if (Lwidth >= Pwidth && (Lheight + Plength) <= B_H) //能装下宽度
444.             return 1;
445.         else
446.             return 0;
447. }
448.
449.
450. //有交配权的所有父代进行交叉
451. void crossover(int& seed)
452. {
453.     const double a = 0.0;
454.     const double b = 1.0;
455.     int mem;
456.     int one;
457.     int first = 0;
458.     double x;
459.
460.     for (mem = 0; mem < GROUP_SCALE; ++mem)
461.     {
462.         x = randT(0.0, 1.0);
463.         //x = r8_uniform_ab(a, b, seed);//产生交配概率
464.
465.         if (x < P_MATING)
```

```
466.         {
467.             ++first;
468.
469.             if (first % 2 == 0)//交配
470.             {
471.                 Xover(one, mem, seed);
472.             }
473.             else
474.             {
475.                 one = mem;
476.             }
477.
478.         }
479.     }
480.     return;
481. }
482.
483. //对最差的染色体和最优的染色体的处理，起到优化的目的
484. void elitist()
485. {
486.     int i;
487.     double best;
488.     int best_mem;
489.     double worst;
490.     int worst_mem;
491.
492.     best = Population[0].Fitness;
493.     worst = Population[0].Fitness;
494.
495.     for (i = 0; i < GROUP_SCALE - 1; ++i)
496.     {
497. #ifdef HighThree
498.
499.         for (int j = TIME-1; j >-1; j--) {
500.             if (Population[i].Fitness >= High[j].Fitness) {
501.                 High[j] = Population[i];
502.                 break;
503.             }
504.         }
505.
506.
507.
508. #endif // HighThree
509.
```

```
510.         if (Population[i + 1].Fitness < Population[i].Fitness)
511.         {
512.
513.             if (best <= Population[i].Fitness)
514.             {
515.                 best = Population[i].Fitness;
516.                 best_mem = i;
517.             }
518.
519.             if (Population[i + 1].Fitness <= worst)
520.             {
521.                 worst = Population[i + 1].Fitness;
522.                 worst_mem = i + 1;
523.             }
524.
525.         }
526.         else
527.         {
528.
529.             if (Population[i].Fitness <= worst)
530.             {
531.                 worst = Population[i].Fitness;
532.                 worst_mem = i;
533.             }
534.
535.             if (best <= Population[i + 1].Fitness)
536.             {
537.                 best = Population[i + 1].Fitness;
538.                 best_mem = i + 1;
539.             }
540.
541.         }
542.
543.     }
544.
545.     //对于当前代的最优值的处理，如果当前的最优值小于上一代则将上一代的值最优个体取代当
    前的最弱个体
546.     //基因保留
547.     if (Population[GROUP_SCALE].Fitness <= best)
548.     {
549.         for (i = 0; i < N_VARS; i++)
550.         {
551.             Population[GROUP_SCALE].Xn[i] = Population[best_mem].Xn[i];
552.         }
```

```
553.         Population[GROUP_SCALE].Fitness = Population[best_mem].Fitness;
554.     }
555.     else
556.     {
557.         for (i = 0; i < N_VARS; i++)
558.         {
559.             Population[worst_mem].Xn[i] = Population[GROUP_SCALE].Xn[i];
560.         }
561.         Population[worst_mem].Fitness = Population[GROUP_SCALE].Fitness;
562.     }
563.     return;
564. }
565.
566.
567.
568.
569.
570. //计算适应度值
571. void evaluate()
572. {
573.     int member;
574.     int i;
575.
576.
577.
578. #ifdef PROFIT
579.     for (member = 0; member < GROUP_SCALE; member++)
580.     {
581.         int sum = Cutting(Population[member].Xn);
582.         double sum_pro = 0;
583.         for (int i = 0; i < sum; i++) {
584.             sum_pro += Profit[(size_t)Population[member].Xn[i]];
585.         }
586.         Population[member].Fitness = sum_pro;
587.     }
588.
589. #else
590.     for (member = 0; member < GROUP_SCALE; member++)
591.     {
592.         int sum = Cutting(Population[member].Xn);
593.         double sum_area = 0;
594.         for (int i = 0; i < sum; i++) {
595.             sum_area += Genetics[(size_t)Population[member].Xn[i]].Area;
596.         }
```

```
597.         Population[member].Fitness = sum_area / B_AREA;
598.     }
599. #endif // PROFIT
600.
601.
602.     return;
603.
604.
605.
606. }
607.
608.
609. //产生整形的随机数
610. int i4_uniform_ab(int a, int b, int& seed)
611. {
612.     int c;
613.     const int i4_huge = 2147483647;
614.     int k;
615.     float r;
616.     int value;
617.
618.     if (seed == 0)
619.     {
620.         cerr << "\n";
621.         cerr << "I4_UNIFORM_AB - Fatal error!\n";
622.         cerr << "  Input value of SEED = 0.\n";
623.         exit(1);
624.     }
625.     //保证 a 小于 b
626.     if (b < a)
627.     {
628.         c = a;
629.         a = b;
630.         b = c;
631.     }
632.
633.     k = seed / 127773;
634.     seed = 16807 * (seed - k * 127773) - k * 2836;
635.
636.     if (seed < 0)
637.     {
638.         seed = seed + i4_huge;
639.     }
640.
```

```
641.     r = (float)(seed) * 4.656612875E-10;
642.     //
643.     //  Scale R to lie between A-0.5 and B+0.5.
644.     //
645.     r = (1.0 - r) * ((float)a - 0.5)
646.         + r * ((float)b + 0.5);
647.     //
648.     //  Use rounding to convert R to an integer between A and B.
649.     //
650.     value = round(r);//四舍五入
651.     //保证取值不越界
652.     if (value < a)
653.     {
654.         value = a;
655.     }
656.     if (b < value)
657.     {
658.         value = b;
659.     }
660.
661.     return value;
662. }
663.
664. //初始化种群个体
665. void initGroup(int& seed)
666.
667. {
668.     int i;
669.     int j;
670.     double lbound;
671.     double ubound;
672.     //
673.     //  initGroup variables within the bounds
674.     //
675.     for (i = 0; i < N_VARS; i++)
676.     {
677.         //input >> lbound >> ubound;
678.
679.         for (j = 0; j < GROUP_SCALE; j++)
680.         {
681.             Population[j].Fitness = 0;
682.             Population[j].ReFitness = 0;
683.             Population[j].SumFitness = 0;
684.             Population[j].Xn[i] = randT(XnRange[i].Lower, XnRange[i].Upper);
```

```
685.             //Population[j].Xn[i] = r8_uniform_ab(XnRange[i].Lower, XnRange[i]
    .Upper, seed);
686.         }
687.     }
688.
689.     return;
690. }
691.
692.
693. //挑选出最大值，保存在种群数组的最后一个位置
694. void selectBest()
695. {
696.     int cur_best;
697.     int mem;
698.     int i;
699.
700.     cur_best = 0;
701.
702.     for (mem = 0; mem < GROUP_SCALE; mem++)
703.     {
704.         if (Population[GROUP_SCALE].Fitness < Population[mem].Fitness)
705.         {
706.             cur_best = mem;
707.             Population[GROUP_SCALE].Fitness = Population[mem].Fitness;
708.         }
709.     }
710.
711.     for (i = 0; i < N_VARS; i++)
712.     {
713.         Population[GROUP_SCALE].Xn[i] = Population[cur_best].Xn[i];
714.     }
715.
716.     return;
717. }
718.
719. //个体变异
720. void mutate(int& seed)
721. {
722.     const double a = 0.0;
723.     const double b = 1.0;
724.     int i;
725.     int j;
726.     double lbound;
727.     double ubound;
```

```
728.     double x;
729. 
730.     for (i = 0; i < GROUP_SCALE; i++)
731.     {
732.         for (j = 0; j < N_VARS; j++)
733.         {
734.             //x = r8_uniform_ab(a, b, seed);
735.             x = randT(a, b);//突变概率
736.             if (x < P_MUTATION)
737.             {
738.                 lbound = XnRange[j].Lower;
739.                 ubound = XnRange[j].Upper;
740.                 Population[i].Xn[j] = randT(lbound, ubound);
741.                 //Population[i].Xn[j] = r8_uniform_ab(lbound, ubound, seed);
742.             }
743.         }
744.     }
745. 
746.     return;
747. }
748. 
749. //模板函数，用于生成各种区间上的数据类型
750. template<typename T>
751. T randT(T Lower, T Upper)
752. {
753.     return rand() / (double)RAND_MAX * (Upper - Lower) + Lower;
754. }
755. 
756. 
757. vector<int>& RandomSeq(){
758. 
759.     vector<int> A;
760.     A.push_back(1);
761.     A.push_back(0);
762.     if (rand() % 2 != 0) {
763.         A[0] = 0;
764.         A[1] = 1;
765.     }
766.     return A;
767. }
768. 
769. 
770. 
771. //产生小数随机数
```

```
772. double r8_uniform_ab(double a, double b, int& seed)
773. 
774. {
775.     int i4_huge = 2147483647;
776.     int k;
777.     double value;
778. 
779.     if (seed == 0)
780.     {
781.         cerr << "\n";
782.         cerr << "R8_UNIFORM_AB - Fatal error!\n";
783.         cerr << "  Input value of SEED = 0.\n";
784.         exit(1);
785.     }
786. 
787.     k = seed / 127773;
788.     seed = 16807 * (seed - k * 127773) - k * 2836;
789. 
790.     if (seed < 0)
791.     {
792.         seed = seed + i4_huge;
793.     }
794. 
795.     value = (double)(seed) * 4.656612875E-10;
796. 
797.     value = a + (b - a) * value;
798. 
799.     return value;
800. }
801. 
802. //输出每一代进化的结果
803. void report(int Xnration)
804. {
805.     double avg;
806.     double best_val;
807.     int i;
808.     double square_sum;
809.     double stddev;
810.     double sum;
811.     double sum_square;
812. 
813.     if (Xnration == 0)
814.     {
815.         cout << "\n";
```

```
816.         cout << "  Xnration       Best            Average       Standard \n";
817.         cout << "  number          value           Fitness       deviation \n
    ";
818.         cout << "\n";
819.     }
820.     sum = 0.0;
821.     sum_square = 0.0;
822.
823.     for (i = 0; i < GROUP_SCALE; i++)
824.     {
825.         sum = sum + Population[i].Fitness;
826.         sum_square = sum_square + Population[i].Fitness * Population[i].Fitnes
    s;
827.     }
828.
829.     avg = sum / (double)GROUP_SCALE;
830.     square_sum = avg * avg * GROUP_SCALE;
831.     stddev = sqrt((sum_square - square_sum) / (GROUP_SCALE - 1));
832.     best_val = Population[GROUP_SCALE].Fitness;
833.
834.     cout << "   " << setw(8) << Xnration
835.         << "   " << setw(14) << best_val
836.         << "   " << setw(14) << avg
837.         << "   " << setw(14) << stddev << "\n";
838.
839.     return;
840. }
841.
842. //选择有交配权的父代
843. void selector(int& seed)
844. {
845.     struct Individual NewPopulation[GROUP_SCALE + 1];//临时存放挑选的后代个体
846.     const double a = 0.0;
847.     const double b = 1.0;
848.     int i;
849.     int j;
850.     int mem;
851.     double p;
852.     double sum;
853.
854.     sum = 0.0;
855.     for (mem = 0; mem < GROUP_SCALE; mem++)
856.     {
```

```
857.         sum = sum + Population[mem].Fitness;
858.     }
859.     //计算概率密度
860.     for (mem = 0; mem < GROUP_SCALE; mem++)
861.     {
862.         Population[mem].ReFitness = Population[mem].Fitness / sum;
863.     }
864.     // 计算累加分布，思想是轮盘法
865.     Population[0].SumFitness = Population[0].ReFitness;
866.     for (mem = 1; mem < GROUP_SCALE; mem++)
867.     {
868.         Population[mem].SumFitness = Population[mem - 1].SumFitness +
869.             Population[mem].ReFitness;
870.     }
871.     // 选择个体为下一代繁殖，选择优秀的可能性大，这是轮盘法的奥秘之处
872.     for (i = 0; i < GROUP_SCALE; i++)
873.     {
874.         p = r8_uniform_ab(a, b, seed);
875.         if (p < Population[0].SumFitness)
876.         {
877.             NewPopulation[i] = Population[0];
878.         }
879.         else
880.         {
881.             for (j = 0; j < GROUP_SCALE; j++)
882.             {
883.                 if (Population[j].SumFitness <= p && p < Population[j + 1].Sum
    Fitness)
884.                 {
885.                     NewPopulation[i] = Population[j + 1];
886.                 }
887.             }
888.         }
889.     }
890.     //更新后代个体
891.     for (i = 0; i < GROUP_SCALE; i++)
892.     {
893.         Population[i] = NewPopulation[i];
894.     }
895.     return;
896. }
897.
898. //显示系统时间
899. void showTime()
```

```
{
# define TIME_SIZE 40

    static char time_buffer[TIME_SIZE];
    const struct tm* tm;
    size_t len;
    time_t now;

    now = time(NULL);
    tm = localtime(&now);

    len = strftime(time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm);

    cout << time_buffer << "\n";

    return;
# undef TIME_SIZE
}

//交叉产生子代
void Xover(int one, int two, int& seed)
{
    int i;
    int point;
    double t;
    //随机选择交叉点，这里的点是以变量的整个长度为单位
    point = randT<int>(0, N_VARS - 1);
    //point = i4_uniform_ab(0, N_VARS - 1, seed);
    //交叉
    for (i = 0; i < point; i++)
    {
        t = Population[one].Xn[i];
        Population[one].Xn[i] = Population[two].Xn[i];
        Population[two].Xn[i] = t;
    }
    return;
}
```