所属类别	2024 年“华数杯”全国大学生数学建模竞赛	参赛编号
本科组		CM2401978

VLSI 电路单元的全局布局构建模型

摘要

电路单元的自动布局是电子设计自动化中的重要环节之一，其旨在矩形布局区域内确定所有电路单元位置，以最小化单元之间总连接线长并避免单元重叠。本文聚焦于自动布局的全局布局构建阶段。

针对问题一，我们建立了与**电路单元连接口坐标**相关的**线长评估模型**，其综合考虑了**半周长线长模型**和**链式线长模型**。利用**梯度下降法**，以与题中给出的 **RSMT** 的绝对差值作为损失值，计算出了**最优加权系数**，得到了**线长综合评估模型**。

针对问题二，我们在题目的基础上建立了**含重叠电路单元区域的单元密度计算模型**，并构建了**基于电路单元的细胞自动机算法**来实现在满足**密度阈值约束**条件下，布局中间状态到分散状态下的转换，并进一步引入**模拟退火算法**，以最小化总连接线长为优化目标，实现了从分散状态到布局较优解的求解过程。综合问题一中的线长综合评估模型，建立了**全局布局构建模型**。

针对问题三，我们利用**边界值检测**的方法检测到了题中给出的单元格布线密度计算模型的问题，并引入**重叠部分包含的电路元件连接口数量**对区域密度进行加权来对模型进行修正。最终得到了**修正后的网格布线密度计算模型**，并将密度与 RGB 元组建立映射关系，将计算结果以**可视化**的形式呈现。

针对问题四，我们引入了**帕累托最优化**的思想来处理**最小化总连接线长**和**最小化网格布线密度的最大值**两个优化目标，并使用**遗传算法**对多优化目标下的全局布局构建模型进行求解。通过将**解空间**中的电路单元位置坐标转化为**二进制编码**，并定义对应的交叉和变异过程，成功使用遗传算法求解了全局布局构建的优化过程。最终得到了**多优化目标的全局布局构建模型**，实现了中间状态在限定条件下转化为全局布局状态的过程。

关键字： 电路设计　全局布局　细胞自动机　模拟退火　帕累托最优化　遗传算法

一、 问题重述

超大规模集成电路（VLSI）将大量电路单元集成于单一芯片，随着设计度的增加，VLSI 的设计已离不开电子设计自动化。电路单元的自动布局是电子设计自动化中的重要环节之一，本题聚焦于自动布局中的全局布局阶段。

问题一要求在给出的 HPWL 和 RSTM 的基础上，构建一个误差更小的线长评估模型，其估计线长与对应 RSMT 的差值尽可能小，并能应用于评估附件一中的总线长。

问题二要求在问题一的基础上，建立单元格密度的计算模型，并在满足密度阈值的前提下，实现将中间状态转化为全局布局状态，使得该布局的总连接线长最小。

问题三要求找出题中给出的布线密度计算模型的问题，并加以修正。运用修正后的单元格布线密度计算模型对单元格密度进行计算，并将结果可视化。

问题四要求除了最小化总连接线长和满足单元密度约束外，尽可能减小网格布线密度的最大值，即在满足多个优化目标的前提下，实现最终的全局布局。

二、 模型假设

1. 假设当单元格内仅包含一个电路单元时不存在密度阈值的限制。

2. 假设在全局布局阶段允许重叠，但应尽可能的避免重叠。

3. 假设重叠区域的布线密度与区域内包含的电路元件连接口数量成正比。

4. 假设题中给出的可活动区域能够实现电路单元的全局布局。

5. 假设电路元件的重叠不影响总连接线长和布线密度的评估。

三、 符号说明

符号	含义
HL	半周长线长
CL	链式线长
$Loss$	线长评估模型损失值
L_{sum}	总连接线长
Se	有效电路单元面积
ρ	含重叠电路单元区域的单元密度
S	电路元件位置集合
Sw	布线区域面积
ρ^l	外接矩形布线密度
ρ^t	网格布线密度

四、 模型的建立与求解

4.1 问题一：与电路单元连接口坐标相关的线长评估模型

问题一中，要求设计一种与**电路单元连接口坐标**相关的**线长评估模型**，其满足：（1）每组估计线长与对应的 RSMT 的差值尽可能小；（2）能应用于评估附件 1 中的总连接线长。**半周长线长模型**对线长的估算简单有效，但对于多连线接口的情形估计偏小。为此，我们引入**链式线长模型**，其适用于布局设计初期，可以快速估算简单线性布线长度，但易过度估计线长。将两种线长评估模型的结果加权相加，建立**线长综合评估模型**。以与 RSMT 的差值总和作为损失值，利用**梯度下降法**最小化损失值，最终求得线长综合评估模型的**最优加权系数**。

4.1.1 半周长线长模型

半周长线长模型（Half-Perimeter Wirelength）是一种简单且常用的估算连接线长度的方法。它通过计算包含所有连接点的矩形的半周长来估算连接线的长度。其计算简单有效，常用于集成电路设计中的全局布局阶段。

对于第 i 组含有连接关系的电路单元，其包含一组电路单元连接口坐标：

$$C_i = \{(x_1^i, y_1^i), (x_2^i, y_2^i), \ldots, (x_n^i, y_n^i)\}$$

其对应的半周长线长 HL_i 为：

$$HL_i = (x_{\text{max}}^i - x_{\text{min}}^i) + (y_{\text{max}}^i - y_{\text{min}}^i) \tag{1}$$

其中：

$$x_{\text{max}}^i = \text{max}(x_1^i, x_2^i, \ldots, x_n^i)$$

$$x_{\text{min}}^i = \text{min}(x_1^i, x_2^i, \ldots, x_n^i)$$

$$y_{\text{max}}^i = \text{max}(y_1^i, y_2^i, \ldots, y_n^i)$$

$$y_{\text{min}}^i = \text{min}(y_1^i, y_2^i, \ldots, y_n^i)$$

半周长线长模型的优点在于其计算效率高，算力要求低，在连接关系较为简单的情况下误差较小，但其对复杂的多连线接口场景的处理能力有限。HPWL 的计算示意图如图 1。

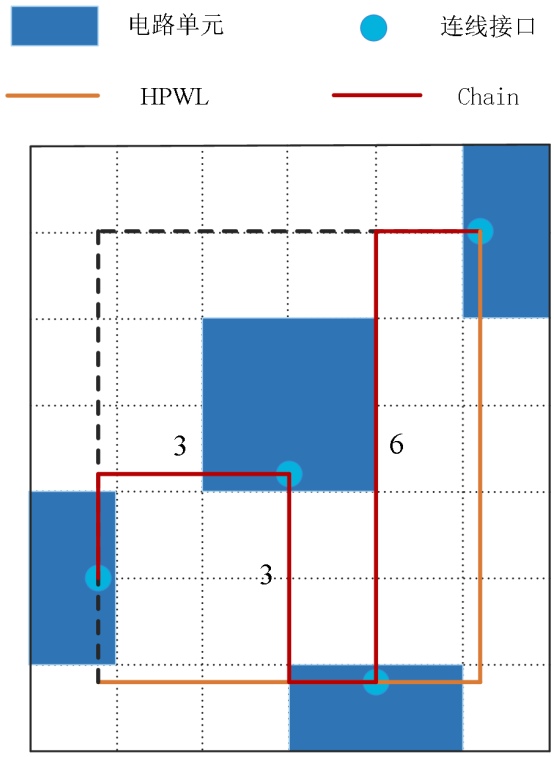

图 1　**HPWL** 和 **Chain** 计算示意图

4.1.2 链式线长模型

链式线长模型（Chain Model）是一种简单的线长评估方法，特别适用于连接点沿线性排列的场景。它通过将所有连接点按顺序连接成一条链来估算总线长。

对于第 i 组含有连接关系的电路单元，其包含一组电路单元连接口坐标：

$$((x_1^i, y_1^i), (x_2^i, y_2^i), \ldots, (x_n^i, y_n^i))$$

其对应的链式线长 CL_i 为：

$$CL_i = \sum_{j=1}^{n-1} \left(|x_{j+1}^i - x_j^i| + |y_{j+1}^i - y_j^i| \right) \tag{2}$$

链式线长模型是一种简单有效的线长评估方法，在特定的应用场景中，链式线长模型是快速评估和优化的有效手段，但需要注意到的是，链式线长模型的评估结果往往偏大。链式线长的计算示意图如图 1。

4.1.3 线长综合评估模型

由于半周长线长模型对线长的评估结果易偏小，而链式线长模型的评估结果往往偏大，因此，我们综合考虑两种线长评估模型，将其加权相加，以实现减小与直线型斯坦纳最小树（RSMT）的绝对误差。考虑线长综合评估模型，其第 i 组含有连接关系的电路单元，其线长评估值 L_i 为：

$$L_i = \alpha \cdot HL_i + (1-\alpha) \cdot CL_i \tag{3}$$

其中 α 为加权系数，通过优化算法确定。

同时，通过线长评估值与直线型斯坦纳最小树给出的最优线长值 R_i 的绝对误差来衡量评估模型在包含 m 组电路单元连接关系数据集上的表现，定义综合评估模型损失值 $Loss$ 为：

$$Loss = \sum_{i=1}^{m} |L_i - R_i|^2 \tag{4}$$

为了更为直观地比较引入链式线长模型对线长评估值准确性的提升，定义半周长线长模型损失值 $Loss^*$ 为：

$$Loss^* = \sum_{i=1}^{m} |HL_i - R_i|^2 \tag{5}$$

并给出包含 m 组电路单元连接关系数据集的总线长 L_{sum}：

$$L_{sum} = \sum_{i=1}^{m} L_i \tag{6}$$

4.1.4 梯度下降法

梯度下降法（Gradient Descent）是一种用于优化目标函数的迭代方法，其通过不断调整模型参数并计算模型梯度，使得目标函数向梯度下降的方向优化，从而最小化目标函数。

对于一个目标函数 $J(\theta)$，其中 θ 是模型参数，GD 的更新公式为：

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t) \tag{7}$$

其中：

- θ_t 是第 t 次迭代的模型参数。

- η 是学习率，控制每次更新的步长。

- $\nabla_\theta J(\theta_t)$ 是目标函数在参数 θ_t 处的梯度。

在本题中，优化目标为由参数 α 唯一决定的评估模型损失值 $Loss(\alpha)$，梯度的计算可以通过与 R_i 的绝对差值来获得，每组具有连接关系的电路元件的 RSMT 在附件中已经给出，因此可以通过梯度下降快速求出在给定附件数据集上的最优参数 α^*，其算法具体流程为：

Algorithm 1 梯度下降算法

```
procedure GradientDescent
    随机初始化加权系数 α 初值。
    repeat
        计算 ∇Loss                          ▷ 目标函数 Loss 在当前参数 α 处的梯度
        α ← α − η∇Loss(α)                   ▷ 利用梯度更新参数
        if ‖∇Loss‖ < ϵ then
            break
        end if
    until 收敛
    return 最优加权系数 α*
end procedure
```

4.1.5 模型求解与灵敏度分析

利用梯度下降法求得最优加权系数 $\alpha^* = 0.8962$，则线长综合评估模型为：

$$L_i = 0.8962Hl_i + 0.1038Cl_i \tag{8}$$

最优加权系数下的线长评估模型损失值 $Loss$ 与半周长线长模型损失值 $Loss^*$ 分别为：

$$Loss = 14289912$$

$$Loss^* = 39805394$$

线长综合评估模型的损失值仅为半周长线长模型损失值的 35.90%，说明引入链式线长评估模型后线长评估的准确度得到了较大程度的提升。进一步的，计算出最优加权

系数下的线长评估模型的总线长为：

$$L_{sum} = 530391$$

4.2 问题二：网格单元密度评估模型与全局布局构建模型

题中给出了不含电路单元重叠区域的单元密度的计算方法，并要求以此为基础给出与电路单元坐标相关的**网格密度评估模型**。同时，应用问题一中构建的线长综合评估模型，整合密度计算，建立**全局布局构建模型**，其满足：（1）最小化总连接线长；（2）满足单元密度约束。由于全局布局阶段允许电路单元的重叠，我们在题目的基础上建立了含重叠电路单元区域的**单元密度计算模型**，并建立了基于电路单元的**细胞自动机算法**。通过对细胞行为进行规则限制，满足了**单元密度阈值** 的限制，并使元件初步分散开来；进一步地，使用**模拟退火算法** 给出使得线长取到最小值的较优解。综合问题一建立的模型，得到了最终的**全局布局构建模型**。

4.2.1 含重叠电路单元区域的单元密度计算模型

电路布线中的单元密度阈值在电路布局设计和布局中起着重要作用。限制单元密度阈值的主要目的，是为了确保电路中的任何区域不会过于拥挤，否则可能会导致电路元件间的信号干扰和串扰、局部过热等问题，并且会增加制造难度。我们认为对单元格内的密度阈值的限制，主要处理的是一个单元格内存在两个及以上的电路元件的情况；同时，有些电路元件较大，无论怎么放置都会至少完整覆盖一个单元格，若此时认为该处单元格的密度为 1，则无法实现对单元格密度阈值的限制。因此，当单元格内不存在电路单元或仅存在一个电路单元时，不考虑其密度阈值的限制，即认为其单元格密度为 0。

当一个单元格内存在两个及以上的电路单元时，在全局布局阶段，允许电路单元之间发生重叠。对于不含电路单元重叠情形的单元密度计算方法，题目中已经给出，由于尽量避免单元重叠也是电路单元自动布局的目标之一，考虑对重叠程度不同的区域加以不同的权重来计算其有效面积，并进一步计算单元密度。这样，在允许重叠的前提下，通过对单元密度阈值的约束，便可实现尽量避免单元重叠的目标。

考虑第 i 个单元格区域 S_i，单元格内存在至少两个电路单元，其面积构成可以表示为：

$$S_i = \{S_0^i, S_1^i, S_2^i, \ldots, S_n^i\}$$

其中 S_n^i 表示该区域为 n 个电路单元共同重叠区域的面积。

定义一个单元格区域内的有效电路单元面积 Se_i：

$$Se_i = \sum_{j=1}^{n} S_j^i \cdot \ln(j + e - 1) \tag{9}$$

当 $j = 1$ 时，该区域的有效电路单元面积计算方式即与题中给出的不含电路单元重叠区域的有效电路单元面积计算方法相同；当该单元格包含电路单元重叠区域时，该区域的有效电路单元面积与重叠单元数量的对数阶成正比。当单元格内出现的重叠电路单元数量增多时，计算得出的有效电路单元面积将迅速增大。

进一步地，定义第 i 个含重叠电路单元区域的单元密度 ρ_i：

$$\rho_i = \frac{Se_i}{\sum_{j=0}^{n} S_j^i} \tag{10}$$

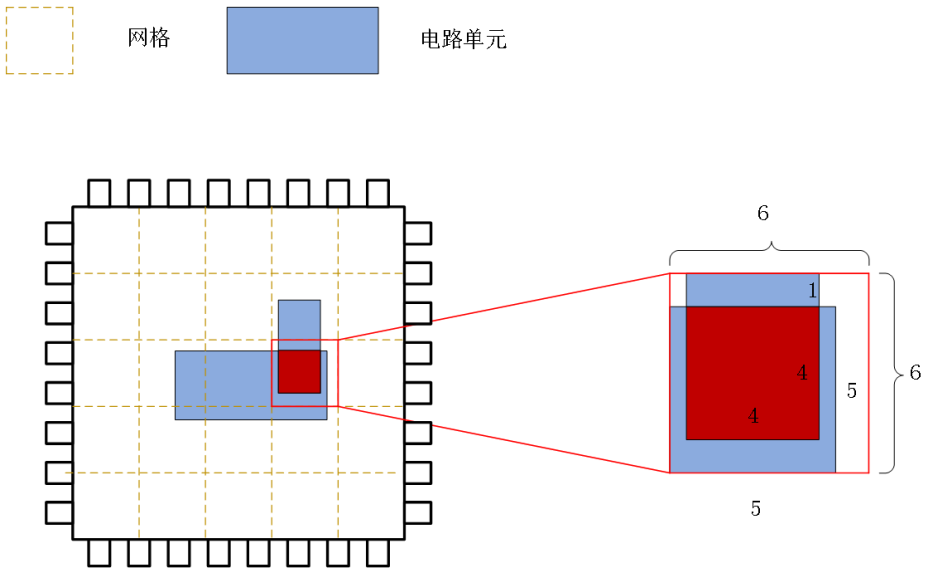

图 **2**　单元密度示意图

以图2为例，若仅以单元格被覆盖面积计算单元密度 ρ_1：

$$\rho_1 = \frac{5 \times 5 + 1 \times 4}{6 \times 6} \approx 0.8056$$

但以含重叠电路单元区域的单元密度计算模型计算单元密度 ρ_2：

$$\rho_2 = \frac{13 \times 1 + 16 \times ln(1+e)}{6 \times 6} \approx 0.9170$$

从数据中看出，当仅以单元格被覆盖面积计算时，该单元格的单元密度小于单元格密度阈值 0.9；但以含重叠电路单元区域的单元密度计算模型计算单元密度时却超出了单元格密度阈值。我们认为，在全局布局阶段，这种程度的单元格利用和电路单元重叠是不可接受的，因为其虽然对该单元格的利用率并不高，但重叠程度过高，将为后续的详细布局带来困难。

4.2.2 基于电路单元的细胞自动机算法

细胞自动机（Cellular Automaton, CA）是一种离散模型，由格子空间、状态集合和局部规则组成，用于模拟复杂系统的演化过程。每个格子被称为一个“细胞”，每个细胞在每个离散时间步都根据一定的规则更新其状态。本题中处理的数据对象是复杂的电路单元分布系统，并且全局布局阶段希望对电路单元的布局进行初步调整，因此可以借鉴细胞自动机的思想来实现这个过程：通过将单元格抽象为“细胞”，通过单元密度来对“细胞”的状态进行界定，并对“细胞”的行为规则加以规定，便可实现让“细胞”整体的状态向指定的趋势发展。

现考虑将电路布局中的每个单元格看做一个“单元格细胞”，定义每个“单元格细胞”的状态为拥挤或者不拥挤，即：

$$\text{Cell-State}_i \in \{\text{Cro}, \text{Unc}\}$$

该状态由该单元格的单元密度决定，即：

$$\text{Cell-State}_i = \begin{cases} \text{Cro}, & \text{if } \rho_i > 0.9 \\ \text{Unc}, & \text{otherwise} \end{cases} \tag{11}$$

同时，定义局部规则，每个“单元格细胞”的行为由周围四个“单元格细胞”的状态和自身的状态共同决定：

（1）若该“单元格细胞”的状态为 Unc，则该“单元格细胞”不进行任何行为；

（2）若该“单元格细胞”的状态为 Cro，且周围的“单元格细胞”的状态均为 Cro，则该“单元格细胞”不进行任何行为；

（3）若该“单元格细胞”的状态为 Cro，且周围的“单元格细胞”至少有一个状态为 Unc，则选取所有状态为 Unc 的“单元格细胞”中单元密度最小的“单元格细胞”，以此单元格细胞与原单元格细胞的相对方向（上、下、左、右）作为移动方向，从有部分区域落于该单元格内的电路元件集合中，随机选取一个电路单元，将其向移动方向整体移动指定步长距离。

为了更直观的观察“单元格细胞”在演化过程中单元格拥挤程度的变化，定义拥挤率 η:

$$\eta = \frac{N_{cro}}{N_{sum}} \tag{12}$$

其中 N_{cro} 为状态为拥挤的“单元格细胞”数，N_{sum} 为所有“单元格细胞”总数。

在定义完局部规则后，通过附件中的数据对“单元格细胞”状态赋予初值，并进行迭代更新，单元格中的电路元件将会在布局范围内逐渐分散开来，通过控制步长来调控分散速度，并限定迭代的终止条件为拥挤率 $\eta = 0$，算法的具体流程示意图如图3。

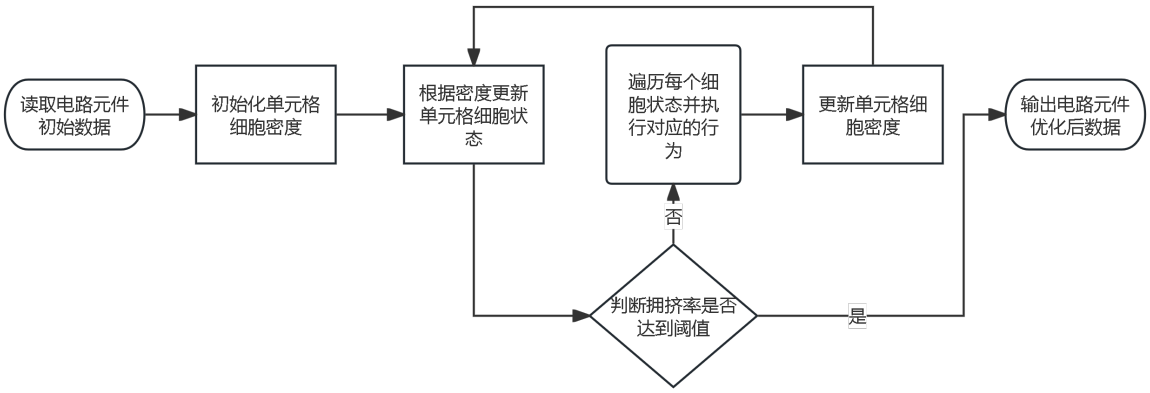

图 **3**　细胞自动机流程图

4.2.3 模拟退火算法

模拟退火算法（Simulated Annealing, SA）是一种用于解决全局优化问题的随机搜索算法。它借鉴了物理退火过程中的基本原理，在一定概率下接受更差的解，以跳出局部最优，逐步趋近全局最优解。模拟退火算法的具体流程为：

Algorithm 2 模拟退火算法

procedure SIMULATEDANNEALING
    随机初始化解 x 和温度 T。
    while T>eps **do**
        计算 x'。 ▷ 在当前解的邻域中随机选择一个新解
        $\Delta E = E(x') - E(x)$。 ▷ 计算目标函数值差
        if $\Delta E < 0$ **then**
            接受新解 x'。
        else
            以概率 $e^{-\Delta E/T}$ 接受新解 x'。
        end if
        减小温度 T。
    end while
return 最优解 x^*。
end procedure

在本题中，目标函数为由线长评估模型、电路元件连接关系集合、电路元件位置集合共同决定的总线长 L_{sum}。其中，线长评估模型由问题一中给出的线长综合评估模型的最优解唯一给出，电路元件连接关系集合由附件信息唯一确定，而电路元件上的连接口在电路元件上的相对位置固定，则唯一影响总线长的变量为电路元件位置集合，即：

$$L_{sum}(S)$$

其中集合 S 为电路元件位置集合：

$$S = \{((x_i, y_i), (x_i + w_i, y_i + h_i)) \mid (x_i, y_i) \in A,\, i \in I\} \tag{13}$$

其中 A 为电路布局区域，I 为所有电路元件矩形左下坐标位置索引集，w_i、h_i 分别为对应电路元件矩形的宽和高。

进一步地，在该集合上定义临域搜索，获取新解 $S\prime$：

$$S\prime = \{((x_i + \delta x_i, y_i + \delta y_i), (x_i + w_i, y_i + h_i)) \mid ((x_j, y_j), (x_j + w_j, y_j + h_j)) \in S\} \quad (14)$$

即从原集合中随机选取部分电路元件，对其电路元件矩形左下角坐标加以微小随机波动 δx_i、δy_i，另外部分电路元件位置信息与原集合保持相同，共同构成新解 $S\prime$。

因此模拟退火算法可在全局范围内对电路元件的位置集合进行小范围调整来寻找总线长的最小值，但需要注意到，在施加微小随机波动时，新解不能超出单元格密度阈值限制，整个模拟退火算法可表示为：

$$SA : (L_{sum}(S))_{min}, \eta(S) = 0 \quad (15)$$

4.2.4 全局布局构建模型

对于附件中给出的全局布局中间状态，我们希望将其进一步转化为全局布局状态，考虑全局布局构建模型，其利用基于电路单元的细胞自动机算法和模拟退火算法，给出一个较优解 S^*，其在允许重叠的前提下尽可能的避免了单元重叠，满足了单元格的密度限制，同时，在一定范围内最小化了总线长。该模型的基本过程为：

（1）利用含重叠电路单元区域的单元密度计算模型来获取解 S_0 的单元格密度；

（2）利用基于电路单元的细胞自动机算法来使较为集中的中间状态分散开来，在这一过程中，减小了电路单元的重叠程度，使得电路单元的分布呈现为较为合理的分散状态 S_1，同时满足了单元密度阈值的约束。

（3）在细胞自动机算法给出的初步状态 S_1 的基础上，使用模拟退火算法进行随机搜索，以最小化总线长为目标，单元密度阈值为限制条件，给出最终全局布局的一个较优解 S^*。全局布局构建模型的示意图如图4。

其特点在于通过层次分析法将题中给出的多个优化目标分步实现，并在分步实现的过程中对解 S 进行逐步优化，模块化的设计降低了求解的复杂程度，同时通过含重叠电路单元区域的单元密度计算模型实现了允许重叠条件下尽量避免重叠的潜在优化目标。全局布局构建模型给出的较优解 S^* 分布合理、总线长较小，较大的减轻了后续详细布

局的工作负担。

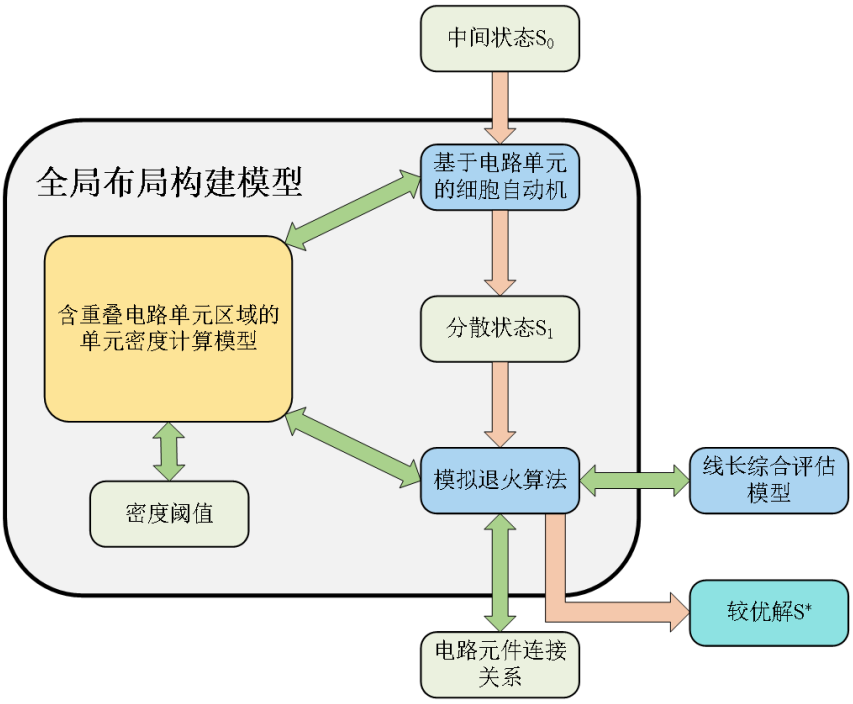

图 **4**　全局布局构建模型示意图

4.2.5 模型求解及应用

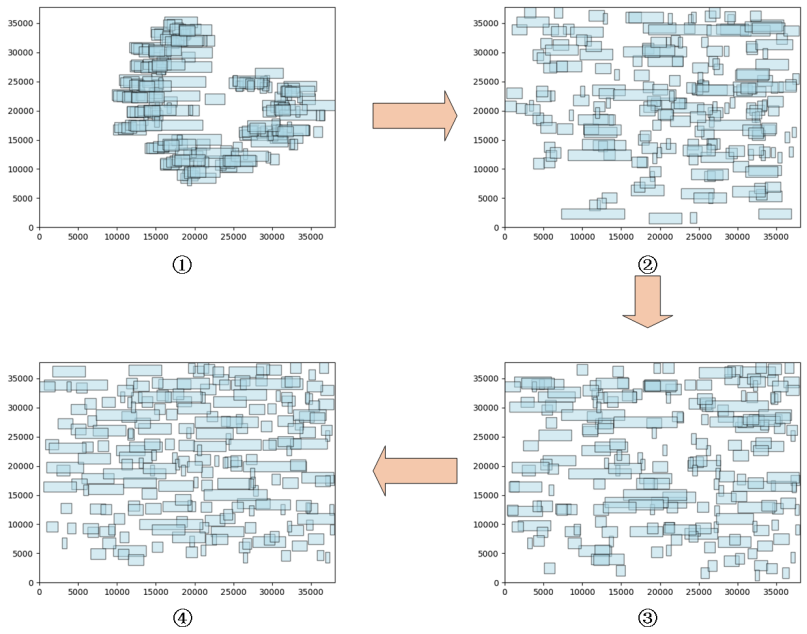

图 **5**　细胞自动机下电路单元分布变化图

由基于电路单元的细胞自动机算法对附件中给出的中间状态进行处理，其在处理过程的不同阶段示意图如图5所示。本文选取了其中具有代表性的四个阶段：初始阶段、初步扩散阶段、扩散至边界阶段、收敛阶段。从图中可以看出电路单元从最初的较为密集的情形逐渐向四周扩散开来，并在接触到边界后进行一定的回收，最终在多次演化后进入到收敛阶段，在该阶段，密度阈值的限制基本满足，电路单元得到了较好的分散。

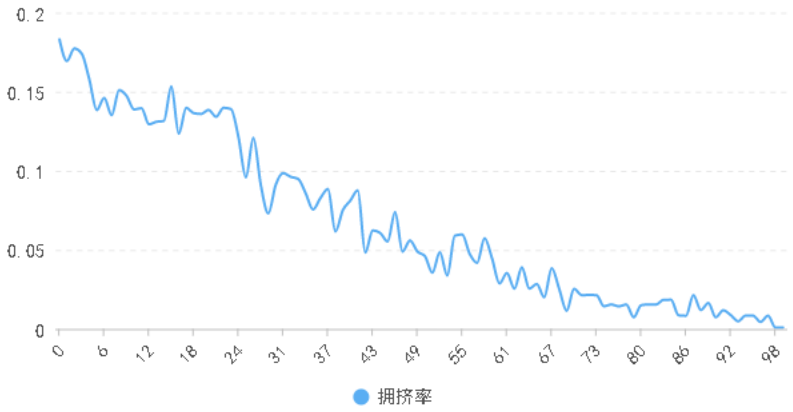

图 **6**　拥挤率变化趋势

在细胞自动机演化的过程中，其拥挤率随迭代轮次变化的曲线图如图6所示。从图中可以看出，拥挤率在迭代前期下降较快，因为前期电路单元较为集中，较小程度的扩散即可较大的降低拥挤率；在迭代后期，电路单元分布基本分散，细胞自动机在多次迭代中逐步缓慢降低拥挤率，最终实现密度阈值的限制，并使电路单元达到分散状态。另外，注意到拥挤率在下降过程中偶尔会出现突然的上升波动，这是因为单元格细胞在移动电路元件时，会出现移动后本单元格细胞仍然为拥挤状态，且移动方向上的单元格细胞由于移动来的电路元件由不拥挤状态也转变为拥挤状态，使得拥挤率在小阶段内呈现上升趋势。但由于移动规则的限制，最终单元格细胞的拥挤率将会逐渐降低并趋向于 0。

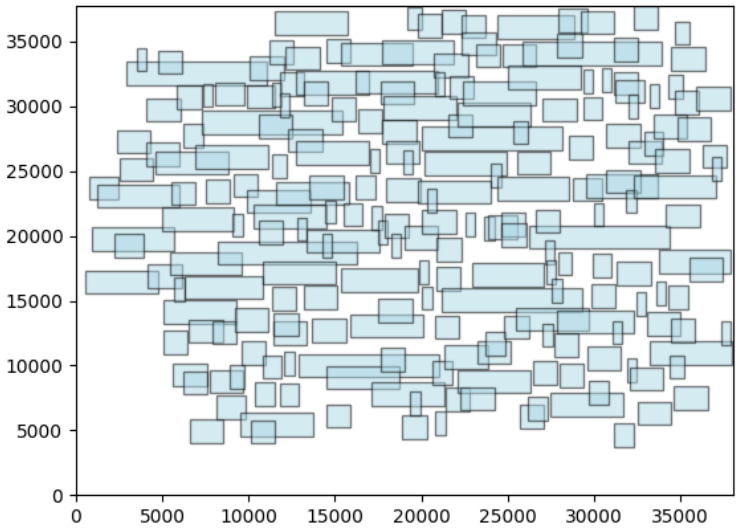

图 **7**　第二问最优解

模拟退火算法对细胞自动机算法给出的分散状态求解得到的较优解分布状态如图7。从图中可以看出，电路单元由分散状态变得更为集中了一些，且重叠程度得到了进一步的缓解。此时的总连接线长 $L_{sum} = 1894180$ ，相较于第一问的总线长有较大程度的增加，这是因为电路元件由中间状态的较为集中情形转变为较优解中的较分散情形，总线长随着电路单元的分散而增加。

4.3 问题三：网格布线密度计算模型

布线密度也是衡量布局质量的重要指标之一。问题二中，题目要求找到给出的网格密度布线密度计算模型存在的问题，对其进行修正，用修正后的计算模型计算出所有单元格的布线密度，并将结果可视化。我们使用**边界值分析**方法，利用特殊情况下计算模型的表现状况找出了模型的问题：未考虑外接矩形内部**电路元件连接口分布不均匀**对局部布线密度的影响。通过引入重叠部分包含的电路元件连接口数量来对区域密度进行加权，得到了**修正后的网格布线密度计算模型**，并将计算结果与 RGB 元组建立映射关系，得到了单元格布线密度的**可视化**结果。

4.3.1 边界值分析

边界值分析（Boundary Value Analysis, BVA）是软件测试中常用的一种方法，其核心思想是通过测试输入边界值附近的值来发现潜在的缺陷，因为程序往往在处理输入边界值时容易出错，边界值分析通过针对这些特殊输入值的测试来提高测试覆盖率和发现缺陷的可能性。其基本步骤为：

1. 确定边界：通常输入或输出等价类的边界就是应该着重测试的边界情况。

2. 选取正好等于、刚刚大于或刚刚小于边界的值作为测试数据，而不是选取等价类中的典型值或任意值。

将边界值分析的软件测试方法运用于本题的问题分析之中，考虑题中给出的网格布线密度计算模型在一些特殊情况、极端情况时计算的正确程度，从而找到给定模型的问题并提出对应的改进方案。

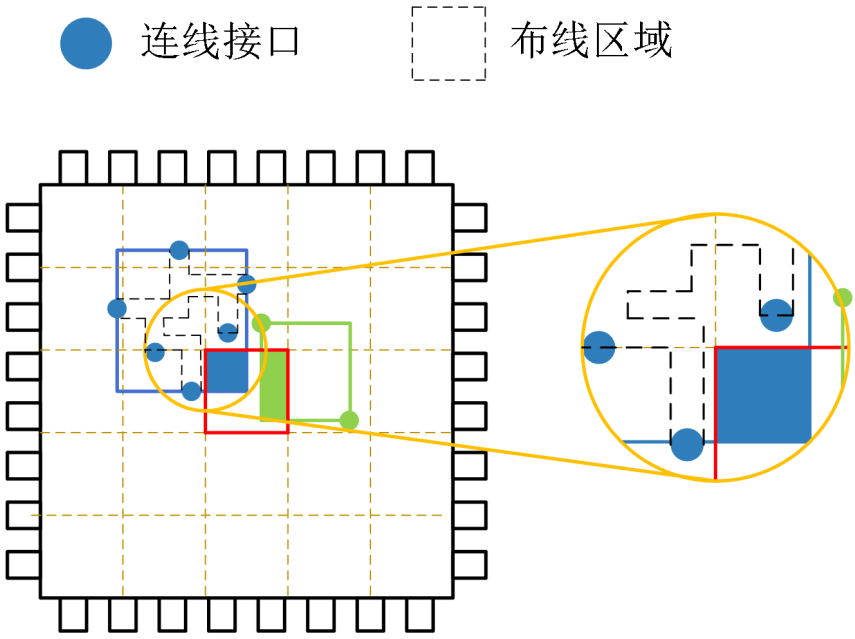

图 **8**　布线密度计算示意图

考察题中给出的示例，其示意图如图8。从图中可以发现，对于这组具有连接关系的电路元件连接口，其给出了一种可行的 HPWL 布线方案，考察外接矩形与单元格的重叠部分，即图中的蓝色区域，该区域内并不存在布线，实际的布线密度应为 0，但按照题中给出的布线密度计算方式，该区域的布线密度被认为与外接矩形整体的布线密度相

同，从而产生了较大误差。

进一步分析可以发现，电路单元连接口在外接矩形内的分布在大部分情况下是不均匀的，因此直接用外界矩形整体的布线密度代替重叠区域的布线密度，就会为单元格的布线密度计算带来误差。进一步考虑极端情况：绝大部分电路单元连接点均集中分布在外接矩形的某一区域，外接矩形与单元格的重叠部分仅有极少数连接点，进而只有极少量连线存在，此时，直接用外界矩形的布线密度代替重叠区域的布线密度，将会为单元格布线密度的计算带来极大误差。因此，题中给出的单元格布线密度计算模型是不可取的，因为其未考虑到矩形内部电路单元连接点分布不均匀的因素。

4.3.2 重叠区域布线密度计算模型

基于题目给出的布线密度计算模型，考虑对重叠区域的布线密度与外接矩形的整体布线密度间的关系进行修正，使其能更准确的表达单元格内的布线密度，具体如下。对于第 i 组含有连接关系的电路单元，其包含 k 个电路单元连接口坐标的集合 C_i：

$$C_i = \{(x_1^i, y_1^i), (x_2^i, y_2^i) \dots (x_k^i, y_k^i)\}$$

根据集合 C_i 可计算出其满足 HPWL 的外接矩形 R_i 和对应的 HPWL 线长：

$$R_i = \{(x_i, y_i), w_i, h_i\}$$

其中 (x_i, y_i) 为该外接矩形 R_i 的左下角坐标，w_i、h_i 为该矩形的宽和长，易知 $HL_i = w_i + h_i$。则布线区域面积 Sw_i 为：

$$Sw_i = HL_i * H_0 \tag{16}$$

其中 H_0 为线宽，题中默认 $H_0 = 1$。进一步的，得到本组单元的总体布线密度 ρ_i^l：

$$\rho_i^l = \frac{Sw_i}{w_i * h_i} \tag{17}$$

考虑与该矩形 R_i 具有重叠部分的单元格 P_j，外接矩形与单元格的重叠部分 M_i^j：

$$M_{(i,j)} = R_i \cap P_j$$

外接矩形内部的电路单元连接口并不是均匀分布的，且 HPWL 给出的布线估计与真实布线方案有一定差距，但想精确的计算出最佳布线方案并求取重叠部分的实际线长，对计算量的要求过高，并不符合实际生产需求，因此我们对重叠部分的区域布线密度进行简化处理，认为其与重叠区域包含的电路单元连接口数量成正相关，对于重叠部分电路单元连接口坐标集合 C_i^*：

$$C_i^* = \{(x_t^i, y_t^i) | (x_t^i, y_t^i) \in M_{(i,j)}\}$$

则重叠部分的电路单元连接口数量 $K_{(i,j)}$ 即为集合 C_i^* 包含的元素数量。对于重叠部分的区域布线密度 $\rho_{(i,j)}^l$：

$$\rho_{(i,j)}^l = \rho_i^l * K_{(i,j)}^{\frac{1}{3}} \tag{18}$$

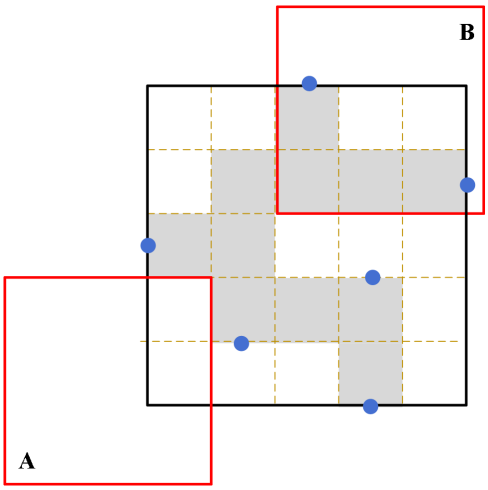

图 **9**　重叠区域布线密度计算示意图

以图9为例，其中蓝点为电路元件连接口，黑色矩形为 HPWL 给出的外接矩形，红色矩形 A、B 为两个单元格，灰色部分为 HPWL 给出的布线方案。计算矩形的布线密度 ρ_0^l:

$$\rho_0^l = \frac{11}{5 \times 5} = 0.44$$

两个单元格 A、B 的布线密度 $\rho_{(0,A)}^l$、$\rho_{(0,B)}^l$ 的真实值分别为：

$$\rho_{(0,A)}^l = \frac{0}{2 \times 1} = 0$$

$$\rho_{(0,B)}^l = \frac{4}{2 \times 3} \approx 0.67$$

此时若直接使用矩形的布线密度 ρ_0^l 来代表单元格的布线密度，则会产生较大误差，若以重叠区域布线密度计算模型来计算单元格的布线密度 $\rho_{(0,A)}^{l*}$、$\rho_{(0,B)}^{l*}$ 为：

$$\rho_{(0,A)}^{l*} = 0.44 \times (0)^{\frac{1}{3}} = 0$$

$$\rho_{(0,B)}^{l*} = 0.44 \times (2)^{\frac{1}{3}} \approx 0.56$$

可以看出重叠区域布线密度计算模型给出的布线密度虽然也有一定误差，但是相较于直接使用外接矩形的布线密度，产生的误差较小，较好的处理了单面格内部电路单元连接口分布不均匀的情况，在绝大多数情形下，连接口越密集的区域，其布线密度相应的越大。将区域包含的连接口数量纳入区域布线密度的计算中，得到的单元格布线密度更均衡、更贴近真实情形。

4.3.3 修正的网格密度计算模型

修正了重叠区域的布线密度与外接矩形的整体布线密度间的关系，进一步可以得出修正的网格密度计算模型，即第 j 个网格的网格布线密度 ρ_j^t 为：

$$\rho_j^t = \frac{\sum_{i=1} \rho_{(i,j)}^l * M_{(i,j)}}{P_j} \tag{19}$$

此处将题中给出的公式进行了进一步的修正，在进行求和之后，应当与单元格的面积作商，保证量纲的正确性。通过修正的网络密度计算模型得到问题二中完成的全局布局下的网格布线密度集合 D^*:

$$D^* = \{\rho_i^* | i \in I^*\}$$

其中 I^* 为单元格索引集合。为了将计算结果以可视化的形式呈现出来，我们将布线密度与 RGB 元组建立对应的映射关系。考虑网格布线密度集合中的最大值与最小值 $\rho_{\max}$、$\rho_{\min}$：

$$\rho_{\max} = \max(D^*)$$

$$\rho_{\min} = \min(D^*)$$

对于每一个布线密度 ρ_i^*，计算其对应的 RGB 值:

$$R_i = 255 \times \frac{\rho_i^* - \rho_{\min}}{\rho_{\max} - \rho_{\min}}$$

$$G_i = 255 \times \left(1 - \frac{\rho_i^* - \rho_{\min}}{\rho_{\max} - \rho_{\min}}\right)$$

$$B_i = 0$$

其可视化结果即映射为 60×64 的热力图。

4.3.4 模型求解

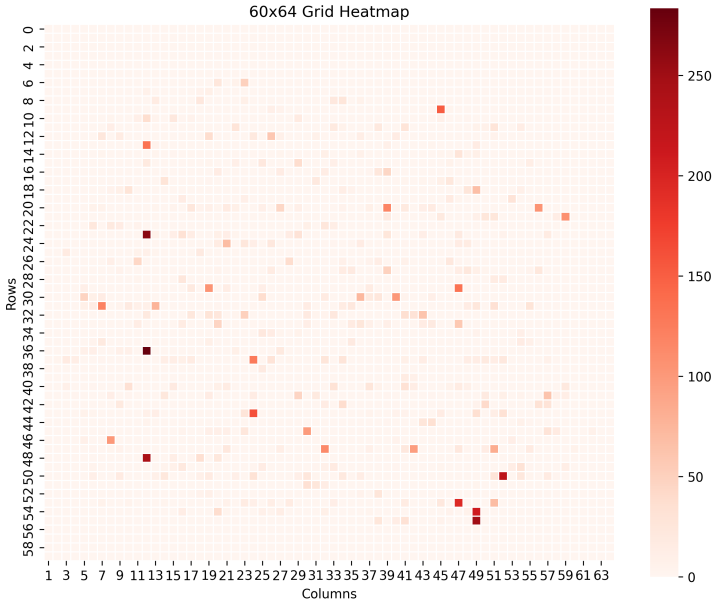

图 **10**　第三问热力图

利用修正后的网格密度布线模型计算问题二中的到的较优解分布状态下的网格布线密度，其结果如图10所示。可以看出，问题二中的较优解的网格密度整体上分布较为均匀，仅存在少数单元格的布线密度较为异常，这可以在后续的详细布局中进一步解决。同时，该可视化图形也可为详细布局处理异常布线提供指导。

4.4 问题四：多优化目标的全局布局构建模型

问题四中，希望除了最小化**总连接线长**和满足单元密度约束外，**网格布线密度**的最大值越小越好。即要求全局布局构建模型在满足密度约束的前提下实现两个优化目

标：（1）最小化总连接线长；（2）最小化网格布线密度的最大值。为此，我们引入**帕累托最优化**思想来处理两个优化目标的多优化问题，并使用**遗传算法**来求解该优化过程，通过将解空间中的电路单元位置坐标转化为二进制编码，并定义对应的交叉和变异过程，实现了全局布局构建模型优化过程到种群繁衍、选择的映射。最终构建了多优化目标的**全局布局构建模型**，并对其进行了求解。

4.4.1 帕累托最优化

帕累托最优化（Pareto Optimization），又称帕累托效率（Pareto Efficiency），是多目标优化中一个重要的概念，其核心思想是找出一个方案，在不损害至少一个目标的情况下改进另一个目标。在本题中，主要的优化目标为两者：（1）最小化总连接线长（2）减小网格布线密度的最大值。两者均由电路单元位置集合 S 唯一决定，即可使用帕累托最优化的思想在优化过程中处理两个优化目标。考虑网格布线密度集合 D^*:

$$D^* = \{\rho_i^* | i \in I^*\}$$

在本题中单元格的数量为 64×60，为了减小计算复杂度，我们将单元格整合为更大的整体区域，每 16 个单元格视为一个大单元格，即密度集合可以表示为：

$$D^* = \{D_1^*, D_2^* \dots D_{240}^*\}$$

对于每一个大单元格区域，其布线密度 ρ_n^{16} 由 16 个小单元格内的布线密度均值来表示：

$$\begin{aligned} \rho_n^{16} &= \frac{1}{16} \sum_{i=1}^{16} \rho_i^* \\ \rho_i^* &\in D_n^* \end{aligned} \tag{20}$$

找到大单元格内的布线密度最大值 ρ_{max}^{16}，其由电路单元位置集合 S 唯一决定：

$$\rho_{max}^{16}(S) = (\rho_n^{16})_{max}$$

则对于多优化目标 $\rho_{max}^{16}(S), L_{sum}(S)$，其一个可接受的帕累托最优解 S_0^* 满足，不存

在另一个解 S_1^* , 使得：

$$\rho_{max}^{16}(S_1^*) \geq \rho_{max}^{16}(S_0^*) \wedge L_{sum}(S_1^*) \geq L_{sum}(S_0^*)$$

$$\rho_{max}^{16}(S_1^*) > \rho_{max}^{16}(S_0^*) \vee L_{sum}(S_1^*) > L_{sum}(S_0^*)$$

4.4.2 遗传算法

遗传算法（Genetic Algorithm，简称 GA）起源于对生物系统所进行的计算机模拟研究，是一种随机全局搜索优化方法，它模拟了自然选择和遗传中发生的复制、交叉和变异等现象，从任一初始种群出发，通过随机选择、交叉和变异操作，产生一群更适合环境的个体，使群体进化到搜索空间中越来越好的区域，这样一代一代不断繁衍进化，最后收敛到一群最适应环境的个体，从而求得问题的优质解。

遗传算法的步骤流程示意图如图11：

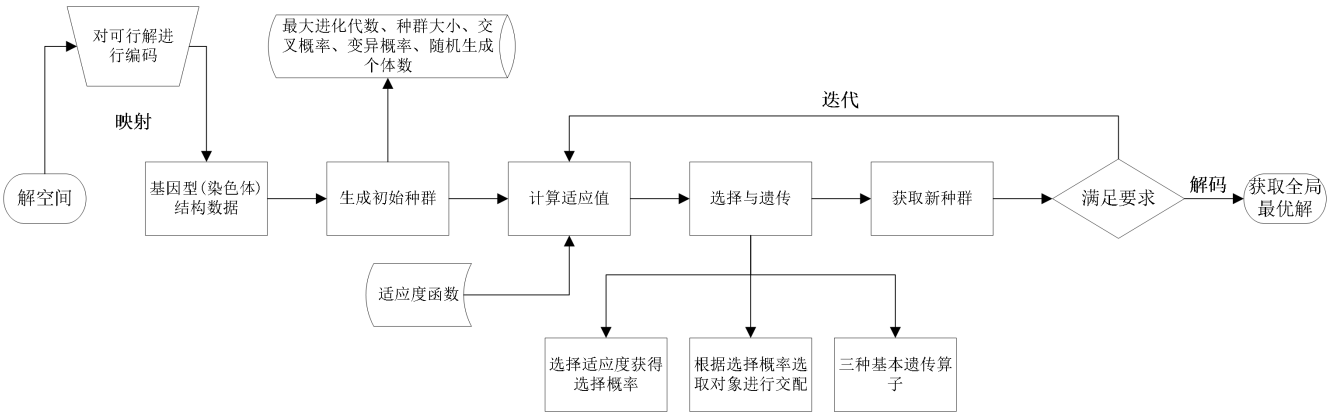

图 **11**　遗传算法流程图

将全局布局构建模型的优化过程抽象为遗传算法模型的关键在于两点：

（1）对可行解进行编码

将解空间 S^* 内的电路单元坐标转化为二进制编码，考虑两个电路单元横坐标转化为的二进制编码串 β_1, β_2，在种群迭代的交叉过程中，两条染色体进行如下操作实现交

叉：

$$\begin{aligned}\beta_1 &= \beta_1 \oplus \beta_2 \\ \beta_2 &= \beta_1 \oplus \beta_2 \\ \beta_1 &= \beta_1 \oplus \beta_2\end{aligned} \tag{21}$$

交叉后的两个电路单元实际上进行了位置的互换。进一步的，考虑变异过程，对一个电路单元横坐标转化为的二进制编码串 β_3，进行如下操作实现变异：

$$\begin{aligned}\beta_3 &= s_1 s_2 \ldots s_{i-1} s_i s_{i+1} \ldots s_n \\ s_i\prime &= 1 - s_i \\ \beta_3\prime &= s_1 s_2 \ldots s_{i-1} s_i\prime s_{i+1} \ldots s_n\end{aligned} \tag{22}$$

变异后的二进制编码串 $\beta_3\prime$ 实际上是对对应的电路单元实现了位置的随机波动。

（2）设计适应度函数

对于通过适应度函数来进行种群筛选，我们利用帕累托最优化的思想，在每轮遗传中，从两个优化目标中按照一定概率选取一个作为主要优化目标，另一个作为次要优化目标，在新种群中选择主要优化目标得到优化的对象，并在这些对象中筛选掉次要优化目标受损的对象。这样，每轮经过筛选的新种群，其主要优化目标得到了优化，而次要优化目标在这轮优化中至少未受损。通过控制优化目标选择时的概率，可以进一步控制遗传算法的主要优化方向。利用适应度函数对种群进行筛选的过程示意图如图12。

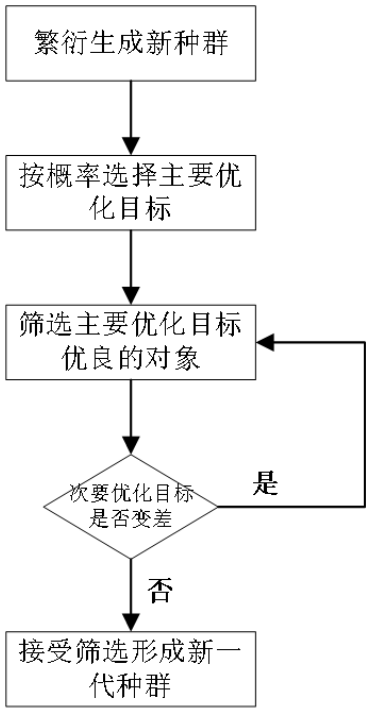

图 **12**　适应度函数筛选流程图

4.4.3 多优化目标的全局布局构建模型与求解

在前文的基础上，我们得到了最终的多优化目标的全局布局构建模型 Build-Model：

$$min \quad Build - Model(L_{sum}(S^*), \rho_{max}^{16}(S^*), p)$$

$$s.t. \quad \eta(S) = 0$$

其中 p 为优化目标选取概率，在本题中 $p = 0.7$，即每次进行选择时有 0.7 的概率选择 $L_{sum}(S^*)$ 作为主要优化目标，整个模型的主要优化趋势为减少总连接线长，在优化结果的基础上，再对布线密度较大的 D_n^* 大单元格内的小单元的电路元件位置进行微调，进一步的降低布线密度，并得到最终的全局布局结果。

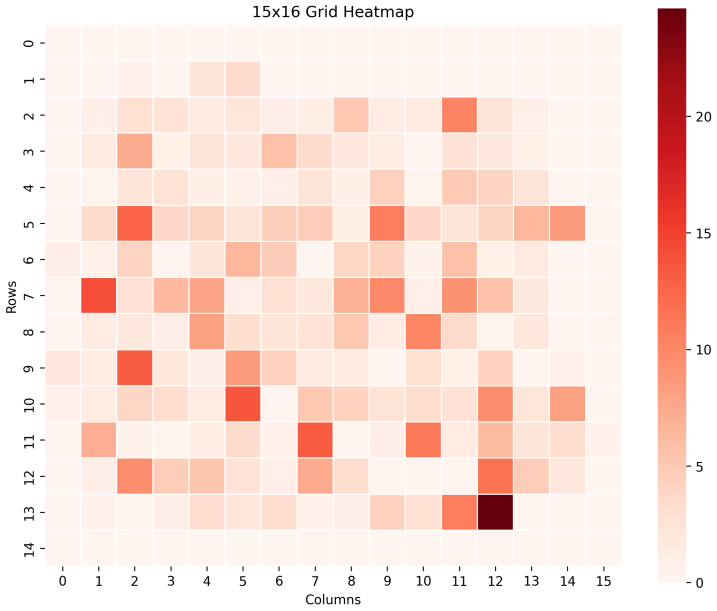

图 **13**　第四问大单元格热力图

图13为遗传算法得出的最优解的大单元布线密度可视化图像，我们对其中布线密度出现异常的大单元内的电路单元位置进行小范围调整，得到最终最优解的布线密度示意图，如图14。其布线密度整体呈现均匀分布，出现线密度较大的单元格数量较问题三进一步减少，这些布线密度较大的单元格则需要在详细布局阶段对电路单元的位置进行更为细致的调整。

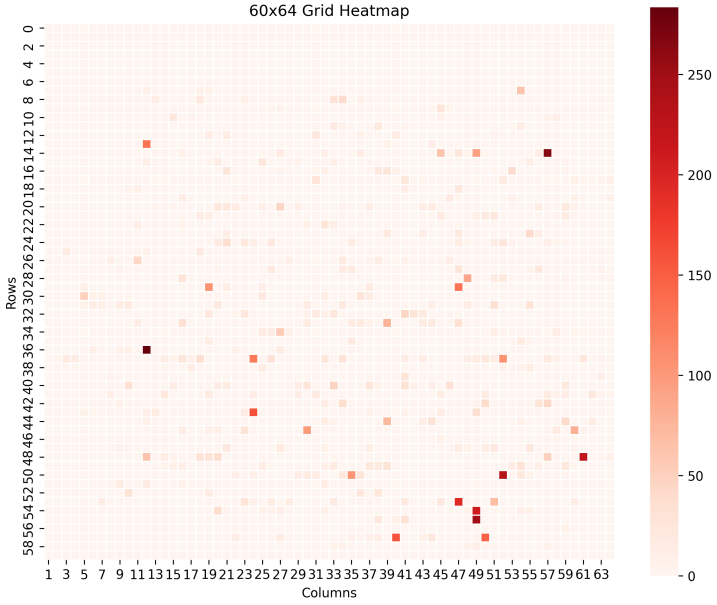

图 **14**　第四问小单元格热力图

该最优解对应的电路单元的布局示意图如图15。从图中可以看出，电路单元进一步集中，且分布更为规则、合理。电路单元的重叠也在可接受范围内，后续的详细布局阶段仅需对电路单元的位置进行微调，即可实现避免重叠，并在满足密度阈值的限制下实现对布局区域的最大化利用。此时的总连接线长 $L_{sum} = 1543721$ ，其在第二问的基础上进一步降低，实现了全局布局阶段下最小化总连接线长、最小化布线密度最大值这两种优化目标，较好的完成了全局布局阶段的任务。

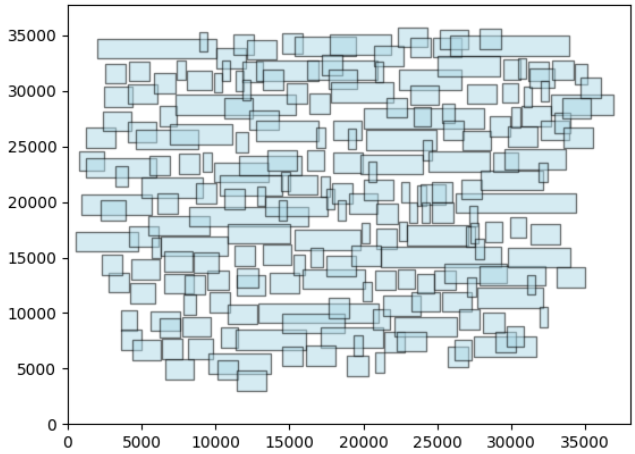

图 **15**　第四问最优解

五、 模型评价

优点：

1. 构建的线长评估模型，其最优加权系数的求解较为简单，进行线长评估时对计算简洁、误差较小。

2. 含重叠区域的单元格密度计算模型充分联系实际，给出的计算标准更为合理，并且通过该模型实现了允许重叠的前提下尽可能的减少重叠的潜在优化目标。

3. 基于本题构建的全新的细胞自动机模型能够有效地处理大规模复杂数据集，其泛化能力强、鲁棒性好。

4. 对于多目标优化问题，将其逐步拆解、一一完成，较好的平衡了多个优化目标之间的关系。

缺点：

1. 最优加权系数对数据集的依赖性较强，对于不同的数据集需要重新计算。

2. 细胞自动机对电路单元的处理较为简单，没有进一步考虑旋转等操作，其解空间相对较小。

3. 遗传算法中的交叉操作较为简单，可以考虑设计更为复杂的交叉操作规则。

推广与应用：

本文中建立的全局布局构建模型可以较好的应用于电路设计中的自动化全局布局，创新性设计的基于电路单元的细胞自动机可以进一步推广和完善，应用于电路设计的不同领域。

参考文献

[1] Caldwell A E, Kahng A B, Markov I L. Design and implementation of move-based heuristics for VLSI hypergraph partitioning[J]. Journal of Experimental Algorithmics (JEA), 2000, 5: 5-es.

[2] 解飞. 一种集成电路全局布局混合整数规划模型 [J]. Operations Research and Fuzziology, 2023, 13: 2878.

[3] 杨之廉, 申明. 超大规模集成电路设计方法学导论 [M]. 清华大学出版社有限公司, 1999.

[4] Markov I L, Hu J, Kim M C. Progress and challenges in VLSI placement research[C]//Proceedings of the International Conference on Computer-Aided Design. 2012: 275-282.

[5] Roy J A, Adya S N, Papa D A, et al. Min-cut floorplacement[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2006, 25(7): 1313-1326.

附录 A　随机梯度下降

```
import numpy as np
import pandas as pd

df_chain = pd.read_excel('Chain.xlsx')
df_hpwl = pd.read_excel('HPWL.xlsx')
df_rsmt = pd.read_excel('RSMT_Values.xlsx')
x = df_chain['Chain Length'].astype(float).values
y = df_hpwl['HPWL'].astype(float).values
z = df_rsmt['RSMT'].astype(float).values
# 计算Hessian矩阵
Hessian = 2 * np.sum((x - y) ** 2)
# 初始参数值p，确保在0和1之间
p = 0.5
n_iterations = 100000
# 损失函数
def loss(p, x, y, z):
   predictions = p * x + (1 - p) * y
   return np.sum((predictions - z) ** 2)
def loss1(x, z):
   predictions = x
   return np.sum((predictions - z) ** 2)
def loss2( y, z):
   predictions =y
   return np.sum((predictions - z) ** 2)
for t in range(n_iterations):
   current_loss = loss(p, x, y, z)
   current_loss1 = loss1(x, z)
   current_loss2 = loss2(y, z)
   print(f"Iteration {t + 1}, Current loss: {current_loss},Current loss1: {current_loss1},
       Current loss2: {current_loss2},Current p: {p}")
   predictions = p * x + (1 - p) * y
   gradient = 2 * np.sum((predictions - z) * (x - y))
   p_new = p - (gradient / Hessian)
   p = np.clip(p_new, 0, 1)
print(f"Optimization complete. Optimal p: {p}, Final loss: {loss(p, x, y, z)}")
```

附录 B　连接线长评估

```
import pandas as pd
def read_excel(file_path):
    return pd.read_excel(file_path, engine='openpyxl')
#计算链式长度
```

```
def calculate_chain_length(points):
    total_length = 0
    for i in range(len(points) - 1):
        total_length += abs(points[i + 1][0] - points[i][0]) + abs(points[i + 1][1] -
            points[i][1])
    return total_length
# 将DataFrame转换为点的列表格式，每个点是一个(x, y)元组
def convert_to_point_list(group_df):
    return list(zip(group_df['X'], group_df['Y']))
# 处理Excel数据，计算每个组的链式长度
def process_data(df):
    results = []
    for group, group_df in df.groupby('Group'):
        points = convert_to_point_list(group_df)
        chain_length = calculate_chain_length(points)
        results.append((group, chain_length))
    return pd.DataFrame(results, columns=['Group', 'Chain Length'])
# 保存结果
def save_to_excel(df, output_file):
    df.to_excel(output_file, index=False, engine='openpyxl')

# 计算HPWL
def calculate_hpwl(group_df):
    # 计算X轴坐标的最大值和最小值
    x_max = group_df['X'].max()
    x_min = group_df['X'].min()
    # 计算Y轴坐标的最大值和最小值
    y_max = group_df['Y'].max()
    y_min = group_df['Y'].min()
    # 计算HPWL，即 x 和 y 差值之和的一半
    hpwl = ((x_max - x_min) + (y_max - y_min))
    return hpwl
```

附录 C　数据处理辅助函数

```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from matplotlib.ticker import MultipleLocator
import pandas as pd


def get_element(document): # 该函数从excel中读取电路元件信息并获得element的初始值
```

```
    """
    从Excel文件中读取电路元件信息，并将每行数据存储为字典。

    参数:
    document (str): Excel文件的路径。

    返回:
    dict: 包含电路元件信息的字典，其中键是行索引，值是包含'x', 'y', 'width', 'height'的字典。
    """
    # 读取Excel文件
    df = pd.read_excel(document)
    # 初始化一个空字典来存储元件信息
    elements = {}
    # 遍历DataFrame中的每一行，创建字典并添加到elements字典中
    for index, row in df.iterrows():
        elements[index] = {
            'x': row['x'],
            'y': row['y'],
            'width': row['width'],
            'height': row['height']
        }
    return elements

def calculate_coverage(elements):
    """
    根据元件字典计算网格覆盖情况。

    参数:
    elements (dict): 元件信息字典，键是索引，值是包含'x', 'y', 'width', 'height'的字典。


    返回:
    np.ndarray: 网格覆盖次数的数组。
    """

    # 设置网格的尺寸和范围
    total_width = 38080
    total_height = 37800
    num_grids_x = 64
    num_grids_y = 60
    grid_size_x = total_width // num_grids_x
    grid_size_y = total_height // num_grids_y

    coverage = np.zeros((num_grids_x * num_grids_y, 16), dtype=int)
    grid_index = 0

    # 遍历每个网格grid_size_x
```

```
for grid_y in range(60):
    for grid_x in range(64):
        # 初始化覆盖次数的二维数组
        coverage_map = np.zeros((grid_size_y, grid_size_x), dtype=int)

        # 定义当前网格的范围
        grid_x_min = grid_x * grid_size_x
        grid_x_max = (grid_x + 1) * grid_size_x
        grid_y_min = grid_y * grid_size_y
        grid_y_max = (grid_y + 1) * grid_size_y

        # 遍历所有元件的字典
        for element_index, element_attrs in elements.items():
            rect_x_min = element_attrs['x']
            rect_y_min = element_attrs['y']
            rect_x_max = rect_x_min + element_attrs['width']
            rect_y_max = rect_y_min + element_attrs['height']

            # 计算矩形与网格重叠区域的坐标
            overlap_x_min = max(grid_x_min, rect_x_min)
            overlap_y_min = max(grid_y_min, rect_y_min)
            overlap_x_max = min(grid_x_max, rect_x_max)
            overlap_y_max = min(grid_y_max, rect_y_max)

            # 检查是否有重叠
            if overlap_x_min < overlap_x_max and overlap_y_min < overlap_y_max:
                # 计算重叠区域的宽度和高度
                overlap_width = overlap_x_max - overlap_x_min
                overlap_height = overlap_y_max - overlap_y_min

                # 计算重叠区域在当前网格中的相对位置和尺寸
                overlap_start_x = (overlap_x_min - grid_x_min)
                overlap_start_y = (overlap_y_min - grid_y_min)
                overlap_end_x = (overlap_x_max - grid_x_min) + 1
                overlap_end_y = (overlap_y_max - grid_y_min) + 1

                # 增加覆盖计数，注意边界条件
                coverage_map[overlap_start_y:overlap_end_y, overlap_start_x:overlap_end_x] += 1

        # 使用 np.unique 获取 coverage_map 中每个不同值的出现次数
        unique_values, counts = np.unique(coverage_map, return_counts=True)

        # 要得到从 0 到 15 的数量，我们可以先初始化一个数组
        coverage_counts = np.zeros(16) # 因为我们需要统计 0 到 15 的数量

        # 然后根据 unique_values 中的值更新 coverage_counts
        for i, count in zip(unique_values, counts):
```

```
            if i < len(coverage_counts): # 确保索引不会超出 coverage_counts 数组的范围
                coverage_counts[i] = count

        coverage[grid_index, :] = coverage_counts
        grid_index = grid_index + 1
    return coverage

def updata_cell_den(element, elements_dict): # 该函数实现读取电路元件信息来更新cell_den
    """

        参数:
        element元件信息字典，键是索引，值是包含'x', 'y', 'width', 'height'的字典。
        elements_dict:调用judege_cell_element的输出: dict:
            键是网格的坐标元组，值是该网格包含的元件序号列表，如果没有元件则为[-1]

        返回:
        64*60数组
            （和我在群里发的地图相比左右相同，但是整体上下颠倒。）（各个网格的相对位置没有问题。）
    """
    coverage = calculate_coverage(element)

    # 将coverage转换为DataFrame
    coverage_df = pd.DataFrame(coverage)

    # 初始化加权求和变量和新的列
    youxiao_sum_series = pd.Series()

    # 遍历DataFrame中的每一行计算youxiao_sum
    for index, row in coverage_df.iterrows():
        weighted_sum = 0
        for col_index, value in enumerate(row):
            if col_index != 0: # 跳过索引为0的列
                weighted_sum += math.log(col_index + math.e - 1) * value
        youxiao_sum = weighted_sum / 374060
        youxiao_sum_series[index] = youxiao_sum

    # 将Series转换为二维NumPy数组，其形状为(60, 64)
    youxiao_sum_array = youxiao_sum_series.values.reshape(60, 64)
    youxiao_sum_array = np.flipud(youxiao_sum_array)

    # 遍历字典
    for (grid_x, grid_y), element_ids in elements_dict.items():
        # 如果元件序号列表的长度等于1
        if len(element_ids) == 1:
            # 设置对应坐标的数组位置为0
            youxiao_sum_array[59 - grid_x, grid_y] = 0
    return youxiao_sum_array
```

```python
def draw_elements(elements): # 该函数实现绘制元件
    # 创建图表
    plt.figure()

    # 设置坐标轴的范围
    plt.xlim(0, 38080) # x轴的范围
    plt.ylim(0, 37800) # y轴的范围

    # 设置x轴和y轴的主刻度间隔
    # ax = plt.gca() # 获取当前的坐标轴对象
    # ax.xaxis.set_major_locator(MultipleLocator(595)) # x轴的主刻度间隔为595
    # ax.yaxis.set_major_locator(MultipleLocator(630)) # y轴的主刻度间隔为630

    # 绘制网格线
    # plt.grid(which='major', color='black', linestyle='-', linewidth=0.5)

    # 绘制矩形
    for index, element in elements.items():
        plt.gca().add_patch(
            Rectangle((element['x'], element['y']), element['width'], element['height'],
                      edgecolor='black',
                      facecolor='lightblue', alpha=0.5))

    # 显示图表
    plt.show()


def save_element(elements, document):
    """
    将元件信息字典保存到Excel文件中。

    参数:
    elements (dict): 元件信息字典，键是索引，值是包含'x', 'y', 'width', 'height'的字典。
    document (str): 要保存的Excel文件路径。
    """
    # 将字典转换为DataFrame
    df = pd.DataFrame.from_dict(elements, orient='index', columns=['x', 'y', 'width', 'height'])

    # 将DataFrame保存到Excel文件
    df.to_excel(document, index=False, header=True, engine='openpyxl')
```

附录 D　细胞自动机

```python
from function import get_element
# 该函数从excel中读取电路元件信息并获得element的初始值
from function import updata_cell_den
# 该函数实现读取电路元件信息来更新cell_den
import random
import matplotlib.pyplot as plt
import numpy as np


def updata_cell_sta(cell_den): # 该函数实现根据cell_den来更新cell_sta
    cell_sta = [[0 for _ in range(64)] for _ in range(60)]
    for i in range(60):
        for j in range(64):
            if cell_den[i][j] > 0.9:
                cell_sta[i][j] = 1
            else:
                cell_sta[i][j] = 0

    return cell_sta


def judege_cell_element(elements):
    """
    判断每个网格包含的元件序号列表。

    参数:
    elements (dict): 元件信息字典，键是索引，值是包含'x', 'y', 'width', 'height'的字典。

    返回:
    dict: 键是网格的坐标元组，值是该网格包含的元件序号列表，如果没有元件则为[-1]。
    """
    total_width = 38080
    total_height = 37800
    num_grids_x = 60
    num_grids_y = 64

    grid_size_x = total_width // 64
    grid_size_y = total_height // 60
    cell_elements = {}

    # 初始化所有网格的元件序号列表为[-1]
    for grid_x in range(num_grids_x):
        for grid_y in range(num_grids_y):
            cell_elements[(grid_x, grid_y)] = [-1]

    # 遍历每个元件
```

```
    for element_index, element in elements.items():
        x, y, width, height = element['x'], element['y'], element['width'], element['height']
        # 检查元件与每个网格的重叠情况
        for grid_x in range(num_grids_x):
            for grid_y in range(num_grids_y):
                grid_x_min = grid_y * grid_size_x
                grid_x_max = grid_x_min + grid_size_x
                grid_y_min = grid_x * grid_size_y
                grid_y_max = grid_y_min + grid_size_y

                # 计算重叠区域
                if grid_x_min < x + width and grid_x_max > x and grid_y_min < y + height and
                    grid_y_max > y:
                    # 如果元件与网格有重叠，则添加元件序号到网格的列表中
                    if element_index not in cell_elements[(grid_x, grid_y)]:
                        cell_elements[(grid_x, grid_y)].append(element_index)
                        if -1 in cell_elements[(grid_x, grid_y)]: # 检查是否需要移除初始的-1
                            cell_elements[(grid_x, grid_y)].remove(-1)

    return cell_elements


document = 'extracted_data.xlsx'
elements = get_element(document) # 是一个字典 key为电路元件的元件序号 value为一个列表
    保存电路元件的左下角横坐标、纵坐标、宽度、高度

cell_index = judege_cell_element(elements) # 是一个字典 key为元组 保存cell的坐标 value为一个列表
    保存该cell内的元件序号

cell_den = updata_cell_den(elements, cell_index) # 是一个60*64的二维列表 用来存储cell的密度
    60个列表 每个列表有64个元素

cell_sta = updata_cell_sta(cell_den) # 一个60*64的二维列表 用来存储cell的状态 0为不拥挤 1为拥挤


# 到这里已经完成了cell_sta、cell_den、elements的初始化


# 细胞自动机行为
def cell_action(cell_sta, elements, cell_den): # 该函数实现根据cell_sta来实现cell的动作
    用于更新element
    elements_index = judege_cell_element(elements) # 获取每个cell包含的元件

    def get_unc_neighbors(i, j):
        neighbors = [] # 用一个列表来存储未拥挤的邻居 该邻居的坐标用元组表示
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        for di, dj in directions:
```

```
            ni, nj = i + di, j + dj
            if 0 <= ni < 60 and 0 <= nj < 64 and cell_sta[ni][nj] == 0:
                neighbors.append((ni, nj))
        return neighbors

    for i in range(60):
        for j in range(64):
            if cell_sta[i][j] == 1: # Cro状态
                unc_neighbors = get_unc_neighbors(i, j)
                elements_move = [] # 用于存储该cell内的元件序号
                if unc_neighbors:
                    # 找到密度最小的Unc单元格并返回其元组坐标
                    min_density_neighbor = min(unc_neighbors, key=lambda pos: cell_den[pos[0]][pos[1]])
                    # 确定移动方向
                    move_direction = (min_density_neighbor[0] - i, min_density_neighbor[1] - j)
                    # 取出元件序列号列表
                    elements_move = elements_index[(59 - i, j)]
                    # 随机选取一个移动元件
                    if elements_move:
                        element_key = random.choice(elements_move)
                        # 移动元件
                        move_x=move_direction[1] * 595 * 10
                        move_y=move_direction[0] * 630 * 6
                        if ((elements[element_key]['x'] + move_x+elements[element_key]['width']) <=
                            38080)and((elements[element_key]['x'] + move_x) >= 0) and
                            ((elements[element_key]['y'] + move_y +elements[element_key]['height']) <=
                            37800)and ((elements[element_key]['y'] + move_y ) >= 0):
                            elements[element_key]['x'] += move_x
                            elements[element_key]['y'] += move_y

def save_element(elements, document):
    """
    将元件信息字典保存到Excel文件中。

    参数:
    elements (dict): 元件信息字典，键是索引，值是包含'x', 'y', 'width', 'height'的字典。
    document (str): 要保存的Excel文件路径。
    """
    # 将字典转换为DataFrame
    df = pd.DataFrame.from_dict(elements, orient='index', columns=['x', 'y', 'width', 'height'])

    # 将DataFrame保存到Excel文件
    df.to_excel(document, index=False, header=True, engine='openpyxl')

from function import draw_elements
import pandas as pd
```

```
# 初始化非拥挤率列表和循环计数器
non_congestion_rates = []
round_counter = 0
max_rounds = 1000

while round_counter < max_rounds:
  #判断是否退出循环
  sum=0
  for i in range(60):
    for j in range(64):
      if cell_sta[i][j] == 0:
        sum+=1
  non_congested_rate = sum / 3840 # 计算非拥挤率
  print(non_congested_rate)

  non_congestion_rates.append(non_congested_rate)



    # 每20轮保存一次非拥挤率
  if (round_counter + 1) % 20 == 0:
    # 将非拥挤率列表写入Excel文件
    with pd.ExcelWriter(f'non_congestion_rates_round_{round_counter + 1}.xlsx') as writer:
      pd.Series(non_congestion_rates).to_excel(writer, sheet_name='NonCongestedRates')

  if sum/3840==1:
    break

  #绘制元件
  if (round_counter + 1) % 10 == 0:
    draw_elements(elements)
  #更新cell_index
  cell_index = judege_cell_element(elements)
  #进行cell的动作 更新elements
  cell_action(cell_sta, elements, cell_den)
  #更新cell_den
  cell_den=updata_cell_den(elements,cell_index)
  #更新cell_sta
  cell_sta=updata_cell_sta(cell_den)

  # 每10轮保存一次元件信息
  if (round_counter + 1) % 10 == 0:
    save_element(elements, f'elements_round_{round_counter + 1}.xlsx')

  # 更新循环计数器
  round_counter += 1
```

```
# 确保最后一轮的非拥挤率也被保存
if round_counter % 20 != 0:
    non_congestion_rates.append(non_congested_rate)
    with pd.ExcelWriter(f'non_congestion_rates_round_{round_counter}.xlsx') as writer:
        pd.Series(non_congestion_rates).to_excel(writer, sheet_name='NonCongestedRates')

# 确保在循环结束后，如果需要，保存最后一次的元件信息
if (round_counter - 1) % 40 == 0:
    save_element(elements, f'elements_final.xlsx')
```

附录 E　模拟退火算法

```
import pandas as pd
import re


def read_sizedata():
    filename='data2.txt'
    dimensions = []
    with open(filename, 'r') as file:
        for line in file:
            parts = line.strip().split(',')
            # 第三个元素是宽度，第四个元素是高度
            width = int(parts[3].strip())
            height = int(parts[4].strip())
            dimensions.append((width, height))
    return dimensions

def parse_data(filename):
    cells_data = {}
    with open(filename, 'r') as file:
        for line in file:
            parts = line.strip().split(',(')
            cell_name = parts[0].strip()
            # 统计逗号的数量
            num_commas = line.count(',')
            num_interfaces=int((num_commas-4)/3)
            # 接口名称和坐标是按顺序对应的
            interfaces_part = parts[2].strip('()') # 移除括号
            interfaces_part_part = interfaces_part.split(',') # 分割接口名称

            for i in range(num_interfaces):
                interfaces = interfaces_part_part[i] # 分割接口名称
                # 接口坐标
```

```
            coordinates_part = parts[3+i].strip('()') # 移除括号
            coordinates_part_part=coordinates_part.split(',')

            coordinates = (int(coordinates_part_part[0]), int(coordinates_part_part[1])) #
                分割并转换坐标

            # 创建接口和坐标的映射
            # 检查 cell_name 是否已经存在于 cells_data 中
            if cell_name not in cells_data:
               # 如果不存在，初始化一个空字典
               cells_data[cell_name] = {}
            cells_data[cell_name][interfaces] = coordinates

    return cells_data

def calculate_absolute_coordinates(parsed_data, cells_coordinates):
    #求每一个接口的绝对坐标。
    #cells_coordinates: 每一个元件的xy坐标的数组。
    # 初始化绝对坐标的字典
    absolute_data = {}
    cells_data = {}
    i=0
    # 遍历 parsed_data 中的每个元件和它的接口
    for cell_name, interfaces in parsed_data.items():

        offset =cells_coordinates[i]
        i = i + 1
        absolute_data = {interface: (x + offset[0], y + offset[1]) for interface, (x, y) in
            interfaces.items()}

        if cell_name not in cells_data:
           # 如果不存在，初始化一个空字典
           cells_data[cell_name] = {}
        cells_data[cell_name] = absolute_data

    return cells_data



def extract_and_print_group_coordinates(data_file, absolute_data):
    # 初始化一个字典来存储每组的连线接口坐标
    group_coordinates = {}

    # 读取并解析data1.txt文件
    with open(data_file, 'r') as file:
        for line in file:
            parts = line.strip().split(',(')
```

```
        group_name = parts[0].strip() # 提取组名称
        connections_str = parts[1].strip('()') # 提取电路单元名称和连线接口名称

        # 使用正则表达式提取电路单元名称和连线接口名称
        connections = re.findall(r'(\w+):(\w+)', connections_str)
        #print('正则表达式提取电路单元名称和连线接口名称',connections)

        # 初始化当前组的坐标列表
        group_coordinates[group_name] = []

        # 遍历每个连接，查找绝对坐标并添加到列表
        for cell, interface in connections:
            if cell in absolute_data and interface in absolute_data[cell]:
                # 获取绝对坐标
                coord = absolute_data[cell][interface]
                # 将绝对坐标添加到当前组的列表
                group_coordinates[group_name].append(coord)
    return group_coordinates


########################优化目标函数#########################
def func(cells_coordinates):
    # cells_coordinates: 每一个元件的xy坐标的数组。

    # 求每一个接口的绝对坐标。
    absolute_coordinates_data = calculate_absolute_coordinates(parsed_data, cells_coordinates)
    # 给出每组接口的绝对坐标。
    data_file = 'data1.txt' # 确保提供正确的文件路径
    group_coordinates = extract_and_print_group_coordinates(data_file, absolute_coordinates_data)
    group_hpwl = calculate_hpwl(group_coordinates)

    group_chain_lengths = calculate_group_chain_lengths(group_coordinates)

    p=0.10380762495494689
    OUTPUT=p*group_chain_lengths+(1-p)*group_hpwl
    return OUTPUT



input_filename = 'data2.txt'
# 输出每个接口相对元件的相对坐标,只计算一次就够。
parsed_data = parse_data(input_filename)


#只提取一次最初始的各个元件的坐标。
# 读取Excel文件
excel_file = '坐标.xlsx'
```

```python
# 使用pandas读取Excel文件，默认读取第一个工作表
df = pd.read_excel(excel_file)
# 提取x和y列的值，并将其转换为整数
cells_coordinates = [(int(row['x']), int(row['y'])) for index, row in df.iterrows()]
print(cells_coordinates)



import math
from random import random
import matplotlib.pyplot as plt
import numpy as np

class SimulateAnnealing:
  def __init__(self, func, initial_solution, iter=10, T0=100, Tf=1e-8, alpha=0.99):
    self.func = func
    self.iter = iter
    self.alpha = alpha
    self.T0 = T0
    self.Tf = Tf
    self.T = T0
    # 假设 initial_solution 是一个形状为 (222, 2) 的数组
    self.solution = [initial_solution.copy() for _ in range(iter)]
    self.history = {'f': [], 'T': []}

  def generate_new(self, current_solution):
    while True:
      # 随机扰动当前解以生成新的解
      new_solution = current_solution + self.T * (np.random.rand(*current_solution.shape) -
          0.5)
      new_solution_limit=sizedata+new_solution
      # 检查 x 坐标是否在 0 到 38080 之间
      x_in_range = (0 <= new_solution_limit[:, 0]) & (new_solution_limit[:, 0] <= 38080)

      # 检查 y 坐标是否在 0 到 37800 之间
      y_in_range = (0 <= new_solution_limit[:, 1]) & (new_solution_limit[:, 1] <= 37800)

      # 检查所有坐标是否都在指定范围内
      all_in_range = x_in_range & y_in_range

      # 限制x和y的坐标在范围内。
      if all(all_in_range):
        break

    return new_solution


  def Metrospolis(self, f, f_new):
```

```
        # 接受新解的逻辑保持不变
        if f_new <= f:
            return 1
        else:
            p = math.exp((f - f_new) / self.T)
            if random() < p:
                return 1
            else:
                return 0

    def get_optimal(self):
        # 找到最优解的逻辑，需要适应多维解
        f_list = [self.func(sol) for sol in self.solution]
        f_best = min(f_list)
        idx = f_list.index(f_best)
        return -f_best, idx, self.solution[idx]

    def plot(self, xlim=None, ylim=None):
        plt.plot(sa.history['T'], sa.history['f'])
        plt.title('Simulate Annealing')
        plt.xlabel('Temperature')
        plt.ylabel('f value')
        if xlim:
            plt.xlim(xlim[0], xlim[-1])
        if ylim:
            plt.ylim(ylim[0], ylim[-1])
        plt.gca().invert_xaxis()
        plt.show()

    def run(self):
        count = 0
        while self.T > self.Tf:
            for i in range(self.iter):
                f = self.func(self.solution[i])
                new_solution = self.generate_new(self.solution[i])
                f_new = self.func(new_solution)
                if self.Metrospolis(f, f_new):
                    self.solution[i] = new_solution
            ft, _, best_solution = self.get_optimal()
            self.history['f'].append(ft)
            self.history['T'].append(self.T)
            self.T *= self.alpha
            count += 1
        print(f"F={ft}, Solution={best_solution}")

sizedata=read_sizedata()
```

```
# initial_solution 是初始解数组
cells_coordinates_array = np.array(cells_coordinates)
initial_solution = cells_coordinates_array
sa = SimulateAnnealing(func, initial_solution)
sa.run()

# plot
sa.plot()
```

附录 F　热力图绘制算法

```
import pandas as pd
import re
import math

def parse_data(filename):
   #输出'Cell1': {'A2': (630, 1051), 'B': (910, 870), 'A1': (385, 672)}（这线口相对坐标）
   cells_data = {}
   with open(filename, 'r') as file:
      for line in file:
         parts = line.strip().split(',(')
         cell_name = parts[0].strip()
         # 统计逗号的数量
         num_commas = line.count(',')
         num_interfaces=int((num_commas-4)/3)
         # 接口名称和坐标是按顺序对应的
         interfaces_part = parts[2].strip('()') # 移除括号
         interfaces_part_part = interfaces_part.split(',') # 分割接口名称

         for i in range(num_interfaces):
            interfaces = interfaces_part_part[i] # 分割接口名称
            # 接口坐标
            coordinates_part = parts[3+i].strip('()') # 移除括号
            coordinates_part_part=coordinates_part.split(',')

            coordinates = (int(coordinates_part_part[0]), int(coordinates_part_part[1])) #
                分割并转换坐标

            # 创建接口和坐标的映射
            # 检查 cell_name 是否已经存在于 cells_data 中
            if cell_name not in cells_data:
               # 如果不存在，初始化一个空字典
               cells_data[cell_name] = {}
            cells_data[cell_name][interfaces] = coordinates
```

```
    return cells_data

def calculate_absolute_coordinates(parsed_data, cells_coordinates):
    #求每一个接口的绝对坐标。

    #parsed_data:parse_data(input_filename)的输出（接口的相对坐标）
    # 'Cell1': {'A2': (630, 1051), 'B': (910, 870), 'A1': (385, 672)}（这线口相对坐标）

    #cells_coordinates: 每一个元件的xy坐标的数组。

    # 初始化绝对坐标的字典
    absolute_data = {}
    cells_data = {}
    i=0
    # 遍历 parsed_data 中的每个元件和它的接口
    for cell_name, interfaces in parsed_data.items():

        offset =cells_coordinates[i]
        i = i + 1
        absolute_data = {interface: (x + offset[0], y + offset[1]) for interface, (x, y) in
            interfaces.items()}

        if cell_name not in cells_data:
            # 如果不存在，初始化一个空字典
            cells_data[cell_name] = {}
        cells_data[cell_name] = absolute_data

    return cells_data

def extract_and_print_group_coordinates(data_file, absolute_data):
    # 初始化一个字典来存储每组的连线接口坐标
    group_coordinates = {}

    # 读取并解析data1.txt文件
    with open(data_file, 'r') as file:
        for line in file:
            parts = line.strip().split(',(')
            group_name = parts[0].strip() # 提取组名称
            connections_str = parts[1].strip('()') # 提取电路单元名称和连线接口名称

            # 使用正则表达式提取电路单元名称和连线接口名称
            connections = re.findall(r'(\w+):(\w+)', connections_str)
            #print('正则表达式提取电路单元名称和连线接口名称',connections)

            # 初始化当前组的坐标列表
            group_coordinates[group_name] = []
```

```
        # 遍历每个连接，查找绝对坐标并添加到列表
        for cell, interface in connections:
            if cell in absolute_data and interface in absolute_data[cell]:
                # 获取绝对坐标
                coord = absolute_data[cell][interface]
                # 将绝对坐标添加到当前组的列表
                group_coordinates[group_name].append(coord)
    return group_coordinates

def calculate_hpwl(group_coordinates):
    # 初始化一个字典来存储每组的HPWL值
    group_hpwl = {}

    # 遍历每组及其坐标
    for group_name, coords in group_coordinates.items():
        # 如果当前组的坐标列表不为空
        if coords:
            # 提取所有x和y坐标
            x_coords = [coord[0] for coord in coords]
            y_coords = [coord[1] for coord in coords]

            # 计算X轴坐标的最大值和最小值
            x_max = max(x_coords)
            x_min = min(x_coords)

            # 计算Y轴坐标的最大值和最小值
            y_max = max(y_coords)
            y_min = min(y_coords)

            # 计算HPWL，即 x 和 y 差值之和的一半
            hpwl = ((x_max - x_min) + (y_max - y_min)) / 2

            # 存储当前组的HPWL值
            group_hpwl[group_name] = hpwl

    return group_hpwl

def calculate_rectangle(group_coordinates):
    # 初始化一个字典来存储每组的矩形面积值
    group_rectangle = {}

    # 遍历每组及其坐标
    for group_name, coords in group_coordinates.items():
        # 如果当前组的坐标列表不为空
        if coords:
            # 提取所有x和y坐标
            x_coords = [coord[0] for coord in coords]
```

```
        y_coords = [coord[1] for coord in coords]

        # 计算X轴坐标的最大值和最小值
        x_max = max(x_coords)
        x_min = min(x_coords)

        # 计算Y轴坐标的最大值和最小值
        y_max = max(y_coords)
        y_min = min(y_coords)

        # 计算rectangle,
        rectangle = ((x_max - x_min) * (y_max - y_min))

        # 存储当前组的HPWL值
        group_rectangle[group_name] = rectangle

    return group_rectangle

#计算外接矩形的密度
def divide_group_values(dict_hpwl, dict_rectangle):
    """
    对两个字典的对应值进行相除操作。

    :param dict_hpwl: 包含HPWL值的字典，键名为组名。
    :param dict_rectangle: 包含矩形面积值的字典，键名为组名。
    :return: 一个新的字典，包含相除的结果。
    """
    # 初始化一个新字典来存储结果
    group_division = {}

    # 遍历字典并执行相除操作
    for key in dict_hpwl:
        # 检查键是否同时存在于两个字典中
        if key in dict_rectangle:
            try:
                # 对应的值相除
                division_result = dict_hpwl[key] / dict_rectangle[key]
                group_division[key] = division_result
            except ZeroDivisionError:
                # 处理除数为0的情况
                print(f"Division by zero occurred for key: {key}")
        else:
            # 如果某个键不存在于另一个字典中，可以选择跳过或者记录日志
            print(f"Key {key} not found in both dictionaries.")

    return group_division
```

```python
def count_connections_in_grid(absolute_coordinates_data):
    total_width = 38080
    total_height = 37800

    cell_width = total_width // 64
    cell_height = total_height // 60

    # 初始化一个字典来存储每个网格位置的连接点信息
    # 这里的值是一个字典，包含组名和对应的点的数量
    grid_dict = {}

    # 遍历每个组的每个连接点
    for group, coords in absolute_coordinates_data.items():
        for coord in coords:
            # 计算连接点所在的网格行列索引
            col_index = coord[0] // cell_width
            row_index = coord[1] // cell_height

            # 构造网格位置的键
            grid_key = (col_index, row_index)

            # 如果该网格位置已经有条目，则更新连接点数量
            if grid_key in grid_dict:
                if group in grid_dict[grid_key]:
                    grid_dict[grid_key][group] += 1
                else:
                    grid_dict[grid_key][group] = 1
            # 如果该网格位置没有条目，创建新条目并初始化连接点数量为1
            else:
                grid_dict[grid_key] = {group: 1}

    # 清理字典，只保留有连接点的网格位置
    # 同时，将每个网格位置中的组名和数量转换为列表形式
    clean_grid_dict = {
        key: {group: count for group, count in value.items()}
        for key, value in grid_dict.items() if value
    }

    return clean_grid_dict

#计算网格中外接矩形的面积
def calculate_rectangle_coverage(absolute_coordinates_data):
    total_width = 38080
    total_height = 37800

    cell_width = total_width // 64
```

```
cell_height = total_height // 60


# 初始化字典来存储每个网格的覆盖情况
grid_coverage = {}

# 遍历每个组及其坐标
for group_name, coords in absolute_coordinates_data.items():
    if not coords:
        continue

    # 提取所有x和y坐标
    x_coords = [coord[0] for coord in coords]
    y_coords = [coord[1] for coord in coords]

    # 计算外接矩形的边界
    x_min, x_max = min(x_coords), max(x_coords)
    y_min, y_max = min(y_coords), max(y_coords)

    # 计算外接矩形覆盖的网格范围
    start_col = max(0, x_min // cell_width)
    end_col = min(64 - 1, (x_max - 1) // cell_width)
    start_row = max(0, y_min // cell_height)
    end_row = min(60 - 1, (y_max - 1) // cell_height)

    # 遍历外接矩形覆盖的所有网格
    for col in range(start_col, end_col + 1):
        for row in range(start_row, end_row + 1):
            # 计算网格的边界
            grid_x_min, grid_x_max = col * cell_width, (col + 1) * cell_width
            grid_y_min, grid_y_max = row * cell_height, (row + 1) * cell_height

            # 计算外接矩形与当前网格的交集
            intersect_x_min = max(grid_x_min, x_min)
            intersect_x_max = min(grid_x_max, x_max)
            intersect_y_min = max(grid_y_min, y_min)
            intersect_y_max = min(grid_y_max, y_max)

            # 检查交集是否有效
            if intersect_x_min < intersect_x_max and intersect_y_min < intersect_y_max:
                intersect_width = intersect_x_max - intersect_x_min
                intersect_height = intersect_y_max - intersect_y_min
                intersect_area = intersect_width * intersect_height

                # 更新网格覆盖字典
                grid_key = (col, row)
                if grid_key not in grid_coverage:
```

```
                grid_coverage[grid_key] = {}
            grid_coverage[grid_key][group_name] = intersect_area

    # 清理字典，删除面积为零的条目
    clean_coverage = {cell: coverage for cell, coverage in grid_coverage.items() if
        any(coverage.values())}

    return clean_coverage

def calculate_grid_product(grid_connection_dict, coverage_dict, group_division):
    # 初始化一个60x64的二维数组，所有值初始化为0
    grid_array = [[0] * 64 for _ in range(60)]

    # 遍历连接点字典，找到每个网格中的组
    for (col, row), groups in grid_connection_dict.items():
        # 确保grid_array的索引不会超出范围
        if row < 60 and col < 64:
            # 遍历该网格中的每个组
            for group, count in groups.items():
                # 如果该组在覆盖字典中有对应的覆盖面积
                if (col, row) in coverage_dict and group in coverage_dict[(col, row)]:
                    area = coverage_dict[(col, row)][group]
                    # 获取该组的密度，如果组不在group_division字典中，则使用默认值1
                    density = group_division.get(group, 1)
                    # 计算乘积（点的数量 * 面积 * 密度）并累加到数组中
                    grid_array[row][col] += math.pow(count,1/3) * area * density

    return grid_array


input_filename = 'data2.txt'
# 输出每个接口相对元件的相对坐标,只计算一次就够。
parsed_data = parse_data(input_filename)

####################读取各个元件的坐标#####################
# 读取Excel文件
excel_file = 'xy.xlsx'
# 使用pandas读取Excel文件，默认读取第一个工作表
df = pd.read_excel(excel_file)
# 提取x和y列的值，并将其转换为整数
cells_coordinates = [(int(row['x']), int(row['y'])) for index, row in df.iterrows()]



# 求每一个接口的绝对坐标。
#cells_coordinates: 每一个元件的xy坐标的数组。
absolute_coordinates_data = calculate_absolute_coordinates(parsed_data, cells_coordinates)
```

```
# 给出每组接口的绝对坐标。
data_file = 'data1.txt' # 确保提供正确的文件路径
group_coordinates = extract_and_print_group_coordinates(data_file, absolute_coordinates_data)



# hpwl
group_hpwl = calculate_hpwl(group_coordinates)

#rectangle
group_rectangle = calculate_rectangle(group_coordinates)

#矩形密度
group_division=divide_group_values(group_hpwl, group_rectangle)

# 统计每个网格中有几个点
# {(38, 16): {'Group1': 1, 'Group134': 1},
grid_connection_dict = count_connections_in_grid(group_coordinates)

#计算网格中外接矩形的面积
coverage_dict = calculate_rectangle_coverage(group_coordinates)

#网格密度
grid_product_array = calculate_grid_product(grid_connection_dict, coverage_dict,group_division)
#print(grid_connection_dict)
#print(coverage_dict)
print(group_division)

import pandas as pd

#保存为Excel表格
def print_grid_to_excel(grid_array, excel_filename='grid_product.xlsx'):
    # 将二维数组转换为DataFrame
    df = pd.DataFrame(grid_array, columns=[f'Cell_{i + 1}' for i in range(64)])

    # 为Excel文件创建一个写入器
    with pd.ExcelWriter(excel_filename, engine='openpyxl') as writer:
        # 将DataFrame写入Excel文件的第一个工作表
        df.to_excel(writer, index=False, sheet_name='Grid Product Array')

    print(f"Grid product array has been written to {excel_filename}")


# 示例使用
# grid_product_array 是之前计算得到的二维数组
print_grid_to_excel(grid_product_array, '网格密度.xlsx')
```

附录 G　遗传算法

```
import numpy as np
import random
import numpy as np

#用于决定本次的优化目标。
def generate_random_bit(probability_one):
    """
    生成一个随机的0或1。

    :param probability_one: 发生1的概率，范围在0到1之间。
    :return: 一个随机整数，0或1。
    """
    return 1 if random.random() < probability_one else 0




import pandas as pd
import re
import math

def parse_data(filename):
    #输出'Cell1': {'A2': (630, 1051), 'B': (910, 870), 'A1': (385, 672)}（这线口相对坐标）
    cells_data = {}
    with open(filename, 'r') as file:
        for line in file:
            parts = line.strip().split(',(')
            cell_name = parts[0].strip()
            # 统计逗号的数量
            num_commas = line.count(',')
            num_interfaces=int((num_commas-4)/3)
            # 接口名称和坐标是按顺序对应的
            interfaces_part = parts[2].strip('()') # 移除括号
            interfaces_part_part = interfaces_part.split(',') # 分割接口名称

            for i in range(num_interfaces):
                interfaces = interfaces_part_part[i] # 分割接口名称
                # 接口坐标
                coordinates_part = parts[3+i].strip('()') # 移除括号
                coordinates_part_part=coordinates_part.split(',')

                coordinates = (int(coordinates_part_part[0]), int(coordinates_part_part[1])) #
                    分割并转换坐标
```

```
            # 创建接口和坐标的映射
            # 检查 cell_name 是否已经存在于 cells_data 中
            if cell_name not in cells_data:
                # 如果不存在，初始化一个空字典
                cells_data[cell_name] = {}
            cells_data[cell_name][interfaces] = coordinates

    return cells_data

def calculate_absolute_coordinates(parsed_data, cells_coordinates):
    #求每一个接口的绝对坐标。

    #parsed_data:parse_data(input_filename)的输出（接口的相对坐标）
    # 'Cell1': {'A2': (630, 1051), 'B': (910, 870), 'A1': (385, 672)}（这线口相对坐标）

    #cells_coordinates: 每一个元件的xy坐标的数组。

    # 初始化绝对坐标的字典
    absolute_data = {}
    cells_data = {}
    i=0
    # 遍历 parsed_data 中的每个元件和它的接口
    for cell_name, interfaces in parsed_data.items():

        offset =cells_coordinates[i]
        i = i + 1
        absolute_data = {interface: (x + offset[0], y + offset[1]) for interface, (x, y) in
            interfaces.items()}

        if cell_name not in cells_data:
            # 如果不存在，初始化一个空字典
            cells_data[cell_name] = {}
        cells_data[cell_name] = absolute_data

    return cells_data

def extract_and_print_group_coordinates(data_file, absolute_data):
    # 初始化一个字典来存储每组的连线接口坐标
    group_coordinates = {}

    # 读取并解析data1.txt文件
    with open(data_file, 'r') as file:
        for line in file:
            parts = line.strip().split(',(')
            group_name = parts[0].strip() # 提取组名称
            connections_str = parts[1].strip('()') # 提取电路单元名称和连线接口名称
```

```
        # 使用正则表达式提取电路单元名称和连线接口名称
        connections = re.findall(r'(\w+):(\w+)', connections_str)
        #print('正则表达式提取电路单元名称和连线接口名称',connections)

        # 初始化当前组的坐标列表
        group_coordinates[group_name] = []

        # 遍历每个连接，查找绝对坐标并添加到列表
        for cell, interface in connections:
            if cell in absolute_data and interface in absolute_data[cell]:
                # 获取绝对坐标
                coord = absolute_data[cell][interface]
                # 将绝对坐标添加到当前组的列表
                group_coordinates[group_name].append(coord)
    return group_coordinates

def calculate_hpwl_dict(group_coordinates):
    # 初始化一个字典来存储每组的HPWL值
    group_hpwl = {}

    # 遍历每组及其坐标
    for group_name, coords in group_coordinates.items():
        # 如果当前组的坐标列表不为空
        if coords:
            # 提取所有x和y坐标
            x_coords = [coord[0] for coord in coords]
            y_coords = [coord[1] for coord in coords]

            # 计算X轴坐标的最大值和最小值
            x_max = max(x_coords)
            x_min = min(x_coords)

            # 计算Y轴坐标的最大值和最小值
            y_max = max(y_coords)
            y_min = min(y_coords)

            # 计算HPWL，即 x 和 y 差值之和的一半
            hpwl = ((x_max - x_min) + (y_max - y_min)) / 2

            # 存储当前组的HPWL值
            group_hpwl[group_name] = hpwl

    return group_hpwl

def calculate_rectangle(group_coordinates):
    # 初始化一个字典来存储每组的矩形面积值
    group_rectangle = {}
```

```
    # 遍历每组及其坐标
    for group_name, coords in group_coordinates.items():
        # 如果当前组的坐标列表不为空
        if coords:
            # 提取所有x和y坐标
            x_coords = [coord[0] for coord in coords]
            y_coords = [coord[1] for coord in coords]

            # 计算X轴坐标的最大值和最小值
            x_max = max(x_coords)
            x_min = min(x_coords)

            # 计算Y轴坐标的最大值和最小值
            y_max = max(y_coords)
            y_min = min(y_coords)

            # 计算rectangle,
            rectangle = ((x_max - x_min) * (y_max - y_min))

            # 存储当前组的HPWL值
            group_rectangle[group_name] = rectangle

    return group_rectangle

#计算外接矩形的密度
def divide_group_values(dict_hpwl, dict_rectangle):
    """
    对两个字典的对应值进行相除操作。

    :param dict_hpwl: 包含HPWL值的字典，键名为组名。
    :param dict_rectangle: 包含矩形面积值的字典，键名为组名。
    :return: 一个新的字典，包含相除的结果。
    """
    # 初始化一个新字典来存储结果
    group_division = {}

    # 遍历字典并执行相除操作
    for key in dict_hpwl:
        # 检查键是否同时存在于两个字典中
        if key in dict_rectangle:
            try:
                # 对应的值相除
                division_result = dict_hpwl[key] / dict_rectangle[key]
                group_division[key] = division_result
            except ZeroDivisionError:
                # 处理除数为0的情况
```

```
            print(f"Division by zero occurred for key: {key}")
        else:
            # 如果某个键不存在于另一个字典中，可以选择跳过或者记录日志
            print(f"Key {key} not found in both dictionaries.")

    return group_division


def count_connections_in_grid(absolute_coordinates_data):
    total_width = 38080
    total_height = 37800

    cell_width = total_width // 64
    cell_height = total_height // 60

    # 初始化一个字典来存储每个网格位置的连接点信息
    # 这里的值是一个字典，包含组名和对应的点的数量
    grid_dict = {}

    # 遍历每个组的每个连接点
    for group, coords in absolute_coordinates_data.items():
        for coord in coords:
            # 计算连接点所在的网格行列索引
            col_index = coord[0] // cell_width
            row_index = coord[1] // cell_height

            # 构造网格位置的键
            grid_key = (col_index, row_index)

            # 如果该网格位置已经有条目，则更新连接点数量
            if grid_key in grid_dict:
                if group in grid_dict[grid_key]:
                    grid_dict[grid_key][group] += 1
                else:
                    grid_dict[grid_key][group] = 1
            # 如果该网格位置没有条目，创建新条目并初始化连接点数量为1
            else:
                grid_dict[grid_key] = {group: 1}

    # 清理字典，只保留有连接点的网格位置
    # 同时，将每个网格位置中的组名和数量转换为列表形式
    clean_grid_dict = {
        key: {group: count for group, count in value.items()}
        for key, value in grid_dict.items() if value
    }

    return clean_grid_dict
```

```python
#计算网格中外接矩形的面积
def calculate_rectangle_coverage(absolute_coordinates_data):
    total_width = 38080
    total_height = 37800

    cell_width = total_width // 64
    cell_height = total_height // 60


    # 初始化字典来存储每个网格的覆盖情况
    grid_coverage = {}

    # 遍历每个组及其坐标
    for group_name, coords in absolute_coordinates_data.items():
        if not coords:
            continue

        # 提取所有x和y坐标
        x_coords = [coord[0] for coord in coords]
        y_coords = [coord[1] for coord in coords]

        # 计算外接矩形的边界
        x_min, x_max = min(x_coords), max(x_coords)
        y_min, y_max = min(y_coords), max(y_coords)

        # 计算外接矩形覆盖的网格范围
        start_col = max(0, x_min // cell_width)
        end_col = min(64 - 1, (x_max - 1) // cell_width)
        start_row = max(0, y_min // cell_height)
        end_row = min(60 - 1, (y_max - 1) // cell_height)

        # 遍历外接矩形覆盖的所有网格
        for col in range(start_col, end_col + 1):
            for row in range(start_row, end_row + 1):
                # 计算网格的边界
                grid_x_min, grid_x_max = col * cell_width, (col + 1) * cell_width
                grid_y_min, grid_y_max = row * cell_height, (row + 1) * cell_height

                # 计算外接矩形与当前网格的交集
                intersect_x_min = max(grid_x_min, x_min)
                intersect_x_max = min(grid_x_max, x_max)
                intersect_y_min = max(grid_y_min, y_min)
                intersect_y_max = min(grid_y_max, y_max)

                # 检查交集是否有效
                if intersect_x_min < intersect_x_max and intersect_y_min < intersect_y_max:
```

```
            intersect_width = intersect_x_max - intersect_x_min
            intersect_height = intersect_y_max - intersect_y_min
            intersect_area = intersect_width * intersect_height

            # 更新网格覆盖字典
            grid_key = (col, row)
            if grid_key not in grid_coverage:
                grid_coverage[grid_key] = {}
            grid_coverage[grid_key][group_name] = intersect_area

    # 清理字典，删除面积为零的条目
    clean_coverage = {cell: coverage for cell, coverage in grid_coverage.items() if
        any(coverage.values())}

    return clean_coverage

def calculate_grid_product(grid_connection_dict, coverage_dict, group_division):
    # 初始化一个60x64的二维数组，所有值初始化为0
    grid_array = [[0] * 64 for _ in range(60)]

    # 遍历连接点字典，找到每个网格中的组
    for (col, row), groups in grid_connection_dict.items():
        # 确保grid_array的索引不会超出范围
        if row < 60 and col < 64:
            # 遍历该网格中的每个组
            for group, count in groups.items():
                # 如果该组在覆盖字典中有对应的覆盖面积
                if (col, row) in coverage_dict and group in coverage_dict[(col, row)]:
                    area = coverage_dict[(col, row)][group]
                    # 获取该组的密度，如果组不在group_division字典中，则使用默认值1
                    density = group_division.get(group, 1)
                    # 计算乘积（点的数量 * 面积 * 密度）并累加到数组中
                    grid_array[row][col] += math.pow(count,1/3) * area * density

    return grid_array

#小网格合并成大网格
def merge_into_large_grid(grid_division):
    """
    将60x64的网格数组合并成15x16的网格数组，每4x4个小网格合并成一个大网格。

    :param grid_division: 原始的60x64的网格数组。
    :return: 合并后的15x16的网格数组。
    """
    # 每个大网格的大小
    merge_size = 4
    # 计算新的行数和列数
```

```
    new_rows = 15
    new_cols = 16

    # 初始化一个15x16的数组，所有值初始化为0
    merged_grid = [[0] * new_cols for _ in range(new_rows)]

    # 遍历每个大网格
    for i in range(new_rows):
        for j in range(new_cols):
            # 计算大网格的均值
            total = 0
            for k in range(merge_size):
                for l in range(merge_size):
                    total += grid_division[i * merge_size + k][j * merge_size + l]
            average = total / (merge_size * merge_size)

            # 存储计算得到的均值到新数组中
            merged_grid[i][j] = average

    return merged_grid

def find_max_value_in_grid(merged_result):
    """
    在15x16的数组中找到数值最大的元素。

    :param merged_result: 15x16的数组。
    :return: 数组中的最大值。
    """
    #转成NumPy数组
    numpy_array = np.array(merged_result)
    # 确保merged_result是NumPy数组
    if not isinstance(numpy_array, np.ndarray):
        raise ValueError("merged_result must be a NumPy array")

    # 使用NumPy的max函数找到最大值
    max_value = np.max(numpy_array)

    return max_value
#########################################################################################################
import pandas as pd
import re

#读取每一个元件的长度和宽度
def read_sizedata():
    filename='data2.txt'
    dimensions = []
    with open(filename, 'r') as file:
```

```
    for line in file:
        parts = line.strip().split(',')
        # 第三个元素是宽度，第四个元素是高度
        width = int(parts[3].strip())
        height = int(parts[4].strip())
        dimensions.append((width, height))
    return dimensions

def parse_data(filename):
    #输出'Cell1': {'A2': (630, 1051), 'B': (910, 870), 'A1': (385, 672)}（这线口相对坐标）
    cells_data = {}
    with open(filename, 'r') as file:
        for line in file:
            parts = line.strip().split(',(')
            cell_name = parts[0].strip()
            # 统计逗号的数量
            num_commas = line.count(',')
            num_interfaces=int((num_commas-4)/3)
            # 接口名称和坐标是按顺序对应的
            interfaces_part = parts[2].strip('()') # 移除括号
            interfaces_part_part = interfaces_part.split(',') # 分割接口名称

            for i in range(num_interfaces):
                interfaces = interfaces_part_part[i] # 分割接口名称
                # 接口坐标
                coordinates_part = parts[3+i].strip('()') # 移除括号
                coordinates_part_part=coordinates_part.split(',')

                coordinates = (int(coordinates_part_part[0]), int(coordinates_part_part[1])) #
                    分割并转换坐标

                # 创建接口和坐标的映射
                # 检查 cell_name 是否已经存在于 cells_data 中
                if cell_name not in cells_data:
                    # 如果不存在，初始化一个空字典
                    cells_data[cell_name] = {}
                cells_data[cell_name][interfaces] = coordinates

    return cells_data

def calculate_absolute_coordinates(parsed_data, cells_coordinates):
    #求每一个接口的绝对坐标。

    #parsed_data:parse_data(input_filename)的输出（接口的相对坐标）
    # 'Cell1': {'A2': (630, 1051), 'B': (910, 870), 'A1': (385, 672)}（这线口相对坐标）

    #cells_coordinates: 每一个元件的xy坐标的数组。
```

```
    # 初始化绝对坐标的字典
    absolute_data = {}
    cells_data = {}
    i=0
    # 遍历 parsed_data 中的每个元件和它的接口
    for cell_name, interfaces in parsed_data.items():

        offset =cells_coordinates[i]
        i = i + 1
        absolute_data = {interface: (x + offset[0], y + offset[1]) for interface, (x, y) in
            interfaces.items()}

        if cell_name not in cells_data:
            # 如果不存在，初始化一个空字典
            cells_data[cell_name] = {}
        cells_data[cell_name] = absolute_data

    return cells_data



def extract_and_print_group_coordinates(data_file, absolute_data):
    # 初始化一个字典来存储每组的连线接口坐标
    group_coordinates = {}

    # 读取并解析data1.txt文件
    with open(data_file, 'r') as file:
        for line in file:
            parts = line.strip().split(',(')
            group_name = parts[0].strip() # 提取组名称
            connections_str = parts[1].strip('()') # 提取电路单元名称和连线接口名称

            # 使用正则表达式提取电路单元名称和连线接口名称
            connections = re.findall(r'(\w+):(\w+)', connections_str)
            #print('正则表达式提取电路单元名称和连线接口名称',connections)

            # 初始化当前组的坐标列表
            group_coordinates[group_name] = []

            # 遍历每个连接，查找绝对坐标并添加到列表
            for cell, interface in connections:
                if cell in absolute_data and interface in absolute_data[cell]:
                    # 获取绝对坐标
                    coord = absolute_data[cell][interface]
                    # 将绝对坐标添加到当前组的列表
                    group_coordinates[group_name].append(coord)
```

```
    return group_coordinates

####计算HPWL值#
def calculate_hpwl(group_coordinates):
    # 初始化HPWL总和
    total_hpwl = 0

    # 遍历每组及其坐标
    for group_name, coords in group_coordinates.items():
        # 如果当前组的坐标列表不为空
        if coords:
            # 提取所有x和y坐标
            x_coords = [coord[0] for coord in coords]
            y_coords = [coord[1] for coord in coords]

            # 计算X轴坐标的最大值和最小值
            x_max = max(x_coords)
            x_min = min(x_coords)

            # 计算Y轴坐标的最大值和最小值
            y_max = max(y_coords)
            y_min = min(y_coords)

            # 计算HPWL，即 x 和 y 差值之和的一半
            hpwl = ((x_max - x_min) + (y_max - y_min)) / 2

            # 累加到HPWL总和
            total_hpwl += hpwl

    return total_hpwl

#计算链式长度
def calculate_chain_length(points):
    total_length = 0
    for i in range(len(points) - 1):
        total_length += abs(points[i + 1][0] - points[i][0]) + abs(points[i + 1][1] -
            points[i][1])
    return total_length

#计算每组链式长度
def calculate_group_chain_lengths(group_coordinates):
    # 初始化链式长度总和
    total_chain_length = 0

    # 遍历每组及其坐标
    for group_name, coords in group_coordinates.items():
        # 对每组的坐标按其x坐标进行排序，确保链式长度计算的连续性
```

```
        sorted_coords = sorted(coords, key=lambda point: point[0])

        # 计算链式长度
        chain_length = calculate_chain_length(sorted_coords)

        # 累加到链式长度总和
        total_chain_length += chain_length

    return total_chain_length

########################优化目标函数#########################
def func(cells_coordinates1):
    cells_coordinates = cells_coordinates1.reshape((222, 2))



    # 求每一个接口的绝对坐标。
    absolute_coordinates_data = calculate_absolute_coordinates(parsed_data, cells_coordinates)
    # 给出每组接口的绝对坐标。
    data_file = 'data1.txt' # 确保提供正确的文件路径
    group_coordinates = extract_and_print_group_coordinates(data_file, absolute_coordinates_data)
    group_hpwl = calculate_hpwl(group_coordinates)

    group_chain_lengths = calculate_group_chain_lengths(group_coordinates)

    p=0.10380762495494689
    OUTPUT=p*group_chain_lengths+(1-p)*group_hpwl
    return OUTPUT

#####################################################################
input_filename = 'data2.txt'
# 输出每个接口相对元件的相对坐标,只计算一次就够。
parsed_data = parse_data(input_filename)

###################读取各个元件的坐标#####################
# 读取Excel文件
excel_file = 'xy.xlsx'
# 使用pandas读取Excel文件，默认读取第一个工作表
df = pd.read_excel(excel_file)
# 提取x和y列的值，并将其转换为整数
cells_coordinates = [(int(row['x']), int(row['y'])) for index, row in df.iterrows()]



#密度的目标函数,输入是222×2的数组元件的xy坐标。
def func_density(cells_coordinates1):
```

```
    cells_coordinates = cells_coordinates1.reshape((222, 2))
    # 求每一个接口的绝对坐标。
    # cells_coordinates: 每一个元件的xy坐标的数组。
    absolute_coordinates_data = calculate_absolute_coordinates(parsed_data, cells_coordinates)
    # 给出每组接口的绝对坐标。
    data_file = 'data1.txt' # 确保提供正确的文件路径
    group_coordinates = extract_and_print_group_coordinates(data_file, absolute_coordinates_data)

    # hpwl
    group_hpwl = calculate_hpwl_dict(group_coordinates)

    # rectangle
    group_rectangle = calculate_rectangle(group_coordinates)

    # 矩形密度
    group_division = divide_group_values(group_hpwl, group_rectangle)

    # 统计每个网格中有几个点
    # {(38, 16): {'Group1': 1, 'Group134': 1},
    grid_connection_dict = count_connections_in_grid(group_coordinates)

    # 计算网格中外接矩形的面积
    coverage_dict = calculate_rectangle_coverage(group_coordinates)

    # 网格密度
    grid_product_array = calculate_grid_product(grid_connection_dict, coverage_dict,
        group_division)
    # 小网格合并成大网格
    merged_result = merge_into_large_grid(grid_product_array)

    max_value = find_max_value_in_grid(merged_result)
    return max_value

#func: 输入222×2的元件坐标输出总线长。
#func_density: 输入222×2的元件坐标输出最大密度

# 优化函数
def func0(array, judge_func):
    if judge_func==1:
        result =func(array)
    else:
        result =func_density(array)
    return result


# 限制条件函数
def func1(array, judge_func):
```

```
    if judge_func == 0:
        result = func(array)
    else:
        result = func_density(array)
    return result



# 遗传算法参数
POPULATION_SIZE = 50 # 种群大小
MUTATION_RATE = 0.01 # 突变率
CROSSOVER_RATE = 0.7 # 交叉率
GENERATIONS = 1 # 迭代次数


# 初始化种群，使用初始数组
def initialize_population(pop_size):
    # 复制初始数组来创建初始种群
    initial_array=cells_coordinates
    population = np.tile(initial_array, (pop_size, 1, 1))
    # 重新整形为 (pop_size, 222*2) 以匹配遗传算法的期望输入
    population = population.reshape(pop_size, -1)
    return population


# 选择过程
def selection(population, fitness):
    idx = np.argsort(fitness)[POPULATION_SIZE:] # 选择适应度最小的POPULATION_SIZE个个体
    return population[idx]


# 交叉过程
def crossover(parent1, parent2):
    crossover_point = np.random.randint(1, len(parent1))
    child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
    child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
    return child1, child2


# 突变过程
def mutate(individual, mutation_rate):
    for i in range(len(individual)):
        if np.random.rand() < mutation_rate:
            individual[i] = np.random.uniform(0, 1)
    return individual
```

```python
# 计算适应度

def calculate_fitness(population, func0, func1, judge_func):
    fitness = []

    for individual in population:
        # 检查 x 坐标是否在 0 到 38080 之间，y 坐标是否在 0 到 37800 之间
        x_coords = individual[::2] # 所有 x 坐标
        y_coords = individual[1::2] # 所有 y 坐标
        if (np.all(x_coords >= 0) and np.all(x_coords <= 38080) and
                np.all(y_coords >= 0) and np.all(y_coords <= 37800)):
            flag=1
        else:
            flag=0

        if func1(individual, judge_func): # 传递 judge_func

            if flag==1:
                fitness.append(func0(individual, judge_func)) # 传递 judge_func
            else:
                fitness.append(np.inf) # 不满足限制条件的适应度设为负无穷

        else:
            fitness.append(np.inf) # 不满足限制条件的适应度设为负无穷
    return np.array(fitness)

# 遗传算法主函数
def genetic_algorithm(func0, func1):
    population = initialize_population(POPULATION_SIZE)

    for generation in range(GENERATIONS):
        judge_func=generate_random_bit(0.8)
        fitness = calculate_fitness(population, func0, func1,judge_func)
        best_idx = np.argmin(fitness) # 找到最佳个体
        best_individual = population[best_idx]
        print(f"Generation {generation}: Best Fitness = {fitness[best_idx]}")

        next_generation = []
        for i in range(0, POPULATION_SIZE, 2):
            parent1, parent2 = population[i], population[(i + 1) % POPULATION_SIZE]
            if np.random.rand() < CROSSOVER_RATE:
                child1, child2 = crossover(parent1, parent2)
                next_generation.extend([child1, child2])
            else:
                next_generation.extend([parent1, parent2])

        for individual in next_generation:
```

```
        mutate(individual, MUTATION_RATE)

    population = next_generation

  return best_individual


# 运行遗传算法
best_array = genetic_algorithm(func0, func1)
print(f"Best Array: {best_array}")
```