中国研究生创新实践系列大赛

中国光谷·“华为杯”第十九届中国研究生数学建模竞赛

学　　校	南京邮电大学
参赛队号	22102930009
队员姓名	1.缪如洋
	2.徐声健
	3.陈凯伦

中国研究生创新实践系列大赛

中国光谷·“华为杯”第十九届中国研究生数学建模竞赛

题 目 PISA 架构芯片资源排布问题

摘 要：

可编程的交换芯片架构 PISA（Protocol Independent Switch Architecture）相较于传统固定性能的交换芯片，其大大提高了研发效率，同时性能不输传统固定性能的交换芯片，有广阔的应用前景。本文针对 PISA 架构的多级报文流水线处理部分提出优化方案，首先对数据进行预处理，得到数据依赖、控制依赖及资源约束查询表，基于贪婪—遗传算法的思想，建立基本块排布优化模型，并针对这一多约束的组合优化问题设计出对应的求解算法，结果表明所设计的算法是有效的，具体问题的解决方案如下所述。

对于问题一，基于贪婪的思想，在最大化本级资源利用率以及最小化整体排布总级数的情况下，考虑数据依赖约束、控制依赖约束及资源约束建立 0-1 整数规划模型，送入遗传算法进行求解，求解出的最优基本块排布级数为 40 级，因为基于贪婪的思想，每一步只考虑当前情况的最优解，故总体的基本块排布从第 0 级到第 39 级呈现逐级递减的态势。另外求解结果也给出了每一级的 4 种资源占用情况并计算每一级的资源利用率，其分布态势与前者类似，各级资源利用率的累加和除以总级数得到总体资源利用率为 39.5%。

对于问题二，在问题一其余情况不变的情况下，另考虑在同一执行流程上的基本块资源占用约束情况，建立 0-1 整数规划模型，送入遗传算法进行求解。最后的结果表明基本块排布级数为 23 级，总体资源利用率为 65.52%。

针对 PISA 架构基本块排布优化的两个问题，我们建立了两个 0-1 整数规划模型，并采用基于启发式算法的贪婪—遗传算法进行模型求解。第一个问求得了最优解，第二个问题在一定时间内求得了满意解。模型有较大的实际应用价值，问题二模型求解的时间复杂度问题还需要进一步研究解决。

关键字：芯片资源排布；0-1 整数规划；贪婪算法；遗传算法

目录

一、问题重述

1.1 问题背景

芯片是现代电子产品不可或缺的核心部件，遍布我们衣食住行的每个角落，而在当前日益复杂的国际形势下，“芯片荒”问题日益凸显，芯片成了各个大国必争的高科技技术。PISA（Protocol Independent Switch Architecture）是当前主流的可编程交换芯片架构之一，其有着和固定功能交换芯片相当的处理速率，同时兼具了可编程性，在未来网络中具有广阔的应用场景[1-2]。本题主要关注 PISA 架构中多级的报文处理流水线部分，在 PISA 架构编程模型中，用户使用 P4 语言描述报文处理行为得到 P4 程序，再由编译器编译 P4 程序，进而生成芯片上可以执行的机器码。编译器在编译 P4 程序时，会首先将 P4 程序划分为一系列的基本块，再将各基本块排布到流水线各级当中。各个基本块会占用一定的资源，而为了减少 PISA 架构芯片设计的复杂度，各流水线级和流水线级之间的资源都有一定的限制，这一系列复杂的资源约束条件使得资源排布问题尤为困难。

PISA 架构芯片资源排布问题指的是将各基本块的资源排布到流水线各级当中，即需要确定每个基本块排布到流水线哪一级使得芯片的资源利用率最高。越高的资源利用率意味着能够越好的发挥芯片的能力，让芯片支持更多的业务，因此，高资源利用率的资源排布算法对于编译器设计至关重要。在芯片资源排布过程中，除了需要考虑各流水线级的资源约束外还需要考虑各基本块之间的数据依赖和控制依赖约束。P4 程序每个基本块均会写一部分变量和读一部分变量，变量的读写使得基本块之间存在数据依赖，同时，基本块执行完后可能跳转到多个基本块执行，从而使得基本块之间也存在着控制依赖，数据依赖和控制依赖约束了基本块排布的流水线级数的大小关系。本问题要解决的就是在满足上述数据依赖、控制依赖、以及各具体子问题的资源约束条件下进行资源排布，使得芯片的资源利用率最大。

1.2 待解决的问题

本题需要根据给定的数据建立芯片资源排布问题的数学模型，并解决以下两个问题：

问题一：给定流水线各级、流水线级与级之间的资源约束条件，以占用的流水线级数尽量短为优化目标，请给出资源排布算法，输出基本块排布结果。

问题二：对于不在一条执行流程上的基本块，可以共享 HASH 资源和 ALU 资源，只要两个基本块中任意一个的 HASH 资源与 ALU 资源均不超过每级资源限制，则两个基本块即可排布到同一级。据此，更改问题一的部分资源约束条件后重新考虑问题 1，给出排布算法，输出基本块排布结果。

二、模型假设

（1）假设基本块的划分是合理的；
（2）假设基本块可以被抽象成一个节点，抽象后基本块中执行的具体指令被屏蔽，只保留读写的变量信息；
（3）假设优化目标占用的流水线级数最短等效于芯片的资源利用率最大；
（4）PISA 架构包括报文解析（parser）、多级的报文处理流水线（Pipeline Pocket Process）、以及报文重组（Deparser）三个组成部分。本题只关注其中的多级报文处理流水线部分，报文解析、报文重组不考虑。
（5）本题已在考虑基本块存在先后执行顺序的基础上对基本块的排布进行数据依赖约束、控制依赖约束及资源依赖约束，故问题 1 中的约束 7：每个基本块只能排布到一级，这一约束条件已然成立。

三、符号说明

表 3-1 符号说明

符号	说明
i	表示基本块的编号
k	表示流水线的级数
P_i	表示 i 基本块所在的流水线级数
A_k	表示位于 k 级流水线的基本块编号的集合
$D_1(i,j)$	表示 i 与 j 是否存在写后读或写后写依赖
$D_2(i,j)$	表示 i 与 j 是否存在读后写依赖
$C(i,j)$	表示 i 与 j 是否存在控制依赖
T_i	表示 i 基本块使用的 TCAM 资源
H_i	表示 i 基本块使用的 HASH 资源
A_i	表示 i 基本块使用的 ALU 资源
Q_i	表示 i 基本块使用的 QUALIFY 资源
Z	表示所有基本块占用的最大流水线级数
N_k	表示偶数级流水线是否占用 TCAM 资源
L_k	表示在 k 级流水线上所有的执行流程
l	表示在一条执行流程上的所有基本块编号

四、问题分析与数据处理

本赛题要求在满足数据依赖、控制依赖和流水线各级资源约束的条件下对基本块进行排布，使其占用的流水线级数最少，从而最大化芯片资源的利用率，提高芯片性能。针对芯片资源排布这一问题，首先需要解决的就是各基本块之间的数据依赖和控制依赖问题，这两种依赖限制了基本块在流水线上级数的先后顺序，只有在满足这两种依赖的情况下，基本块的排布才被视为是有效的。因此，理清各基本块之间的数据依赖和控制依赖关系对于解决问题一和问题二是至关重要的。赛题总共提供了3张基本块信息的表，它们分别是各基本块使用的资源信息表、各基本块读写的变量信息表、各基本块在流程图中的邻接基本块信息表，表中共有607个基本块的信息，在提取基本块之间的数据依赖和控制依赖关系时，主要关注后两张表，第一张表则用作计算流水线级基本块总的资源占用量是否满足子问题的资源约束限制。

4.1 邻接基本块信息表的处理

邻接基本块信息表给出了各基本块与具有有向边连接的其他基本块信息，即一个基本块执行结束后跳转的基本块，通过该表可以得到各基本块的所有后向基本块信息表，也即得到了基本块的执行顺序。表4-1给出了得到的基本块之间的执行顺序信息，其中第一列为基本块编号，后续为执行顺序在第一列基本块之后的所有基本块编号。

表4-1 执行顺序表（部分）

Block	后续基本块编号					
23	24	25	26	57	86	……
24	25	26	57	86	87	……
25	26	86	87	88	89	……
26	86	87	88	89	90	……

图4-1展示了23号基本块所有的后向连接，图中圆圈里的数字代表基本块编号，由此可以得到23号基本块与所有在它执行顺序之后的基本块的控制依赖关系，相关计算方法在下一小节中给出。

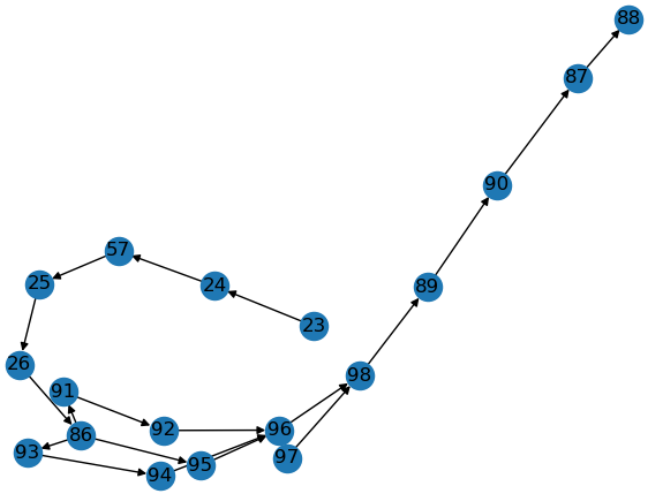

图 4-1 23 号基本块的所有后向连接图

4.2 控制依赖关系的计算

控制依赖是程序控制流导致的一种约束。而控制依赖分析是用来确定一条程序语句语义的变化是否会影响其他程序语句的执行，是程序并行化的基础[3]。

控制依赖定义为：当从某个基本块出发的路径，只有部分路径通过下游某个基本块时，两基本块构成控制依赖。其在程序中的典型表现是条件判断语句，条件判断语句中判断值的真假来决定下一步执行的程序语句。所以存在控制依赖的两个基本块必须要满足先执行的基本块排布的流水线级数小于或等于后执行的基本块排布的流水线级数。

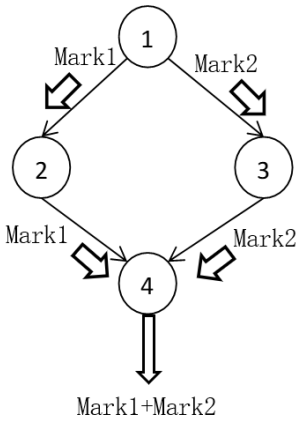

图 4-2 控制依赖关系计算

通过上一节得到的基本块执行顺序表和流程图可以计算出基本块之间的控制依赖关系。由控制依赖定义可知，当一个基本块没有下一步执行的基本块或只有一个邻接基本块时则它与后续所有基本块均无控制依赖关系，所以在 attachment3 表中可以先筛选掉这一部分基本块。而对于有多条有向边连接的基本块则采取标记法来计算后续基本块是否与其有控制依赖关系。其基本思路如图 4-2 所示，对于 1 号基本块来说，其后向邻

接基本块有 2 号和 3 号两个，则对于这两条有向边分别赋予标记“Mark1”和“Mark2”，分别传递给 2 号基本块和 3 号基本块，2 号和 3 号再将它们收到的标记传递到后续基本块，则 4 号得到的标记即为“Mark1+Mark2”包含了 1 号基本块所有出发路径带有的标记，这意味着从 1 号基本块开始的所有路径最后都经过了 4 号基本块，所以 1 号基本块与 4 号基本块之间没有控制依赖。而 2 号和 3 号基本块只含有 1 号所有出发路径标记的一部分，表示从 1 号基本块出发只有部分路径到达 2 号和 3 号，所以存在控制依赖关系，符合控制依赖的定义。

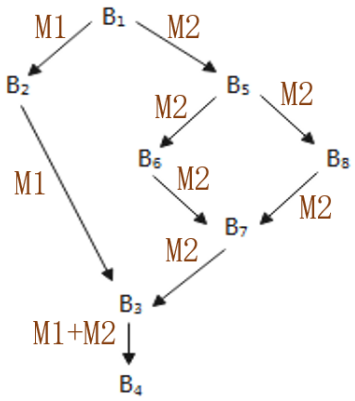

图 4-3 控制依赖关系计算

上图展示了使用标记法计算赛题示例的控制依赖关系，可以看出 B_3 与 B_4 带有所有 B_1 出发路径的标记，所以判定为与 B_1 无控制依赖关系。而 B_2、B_5、B_6、B_7、B_8 均只带有一种标记，所以判定为与 B_1 有控制依赖关系，符合赛题描述。

使用该方法对 607 个基本块都进行计算，便可得出每一个基本块与所有后续基本块是否具有控制依赖关系。通过上一节得到的执行顺序表可以快速地得出控制依赖关系表，如下表所示。

表 4-2 控制依赖关系表（部分）

Block	具有控制依赖关系的基本块编号					
233	5	6	7	8	9	……
234	5	6	7	8	9	……
235	5	6	7	8	9	……
236	0	1	2	5	6	……

4.3 数据依赖关系的计算

根据表 4-1 提供的各基本块之间的执行顺序，结合提供的数据：attachment2.csv 文件，可以确定出已存在先后顺序的两个基本块之间是否满足数据依赖，具体流程如下：

1. 确定任意两个 block 之间的执行顺序，生成先后顺序的查询表：先后顺序.csv。
2. 在存在先后执行顺序的两个 block 之间判定是否存在数据依赖：
   - 前一个 block 存在写操作的变量集合与后一个 block 存在写操作的变量集合存在交集，判定为存在写后写依赖，记为 A；
   - 前一个 block 存在写操作的变量集合与后一个 block 存在读操作的变量集合

存在交集，判定为存在写后读依赖，记为 B；

- 前一个 block 存在读操作的变量集合与后一个 block 存在写操作的变量集合存在交集，判定为存在读后写依赖，记为 C；
- 以上 3 种条件均不满足时，表示不存在数据依赖，记为 D；
- 根据上述规则，生成表格:data_depend2.csv(见附录)。

3. 将得到的表格进行整理：
   - 写后写依赖与写后读依赖都要求前一个 block 级数小于后一个 block 级数，因此归为同一类，记为-1；
   - 读后写依赖要求前一个 block 级数小于等于后一个 block 级数，记为 1；
   - 不存在数据依赖的情况记为 0。
4. 根据上述规则，生成如下表 4-3(截取自附录文件 data_depend3.csv)。

表 4-3 数据依赖关系表（部分）

	0	1	2	3	4	5	6	7	8
27	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0
29	0	0	0			-1	0	-1	-1
30	0	0	0			0	0	0	0
31	0	0	0		-1				
32	0	0	0		0				

五、问题的建模与求解

5.1 问题一的建模与求解

5.1.1 建模准备

问题一要求在给定的资源约束条件下，并且满足控制依赖和数据依赖关系的情况下排布 607 个基本块使得占用的流水线级数数量最小。其中资源约束条件主要包括每一级四种资源的最大数目、0 到 31 级里折叠级数的资源数量限制和含有 TCAM 资源的级数数量限制。在建立模型之前，需要先将各基本块之间的读写数据依赖和控制依赖整理好，可以简化模型的表达，便于后续的求解。

5.1.2 问题一模型的建立

基于问题一的要求，本文建立了一个单目标多约束的芯片资源排布模型。

（1）建立目标函数

本文首先定义以下符号：

i 表示基本块的编号，$i \in [0,606]$；

k 表示流水线的级数；

P_i 表示 i 基本块所在的流水线级数；

A_k 表示位于 k 级流水线的基本块编号的集合：

$$A_k = \{i | P_i = k\}, i \in [0,606] \tag{5-1}$$

$D_1(i, j)$表示 i 号基本块与 j 号基本块是否存在写后读依赖或写后写依赖：

$$D_1(i,j) = \begin{cases} 1, & i与j存在写后读或写后写依赖 \\ 0, & i与j不存在写后读或写后写依赖 \end{cases} \tag{5-2}$$

$D_2(i, j)$表示 i 号基本块与 j 号基本块是否存在读后写依赖：

$$D_2(i,j) = \begin{cases} 1, & i与j存在读后写依赖 \\ 0, & i与j不存在读后写依赖 \end{cases} \tag{5-3}$$

$C(i, j)$表示 i 号基本块与 j 号基本块是否存在控制依赖：

$$C(i,j) = \begin{cases} 1, & i与j存在控制依赖 \\ 0, & i与j不存在控制依赖 \end{cases} \tag{5-4}$$

T_i 表示 i 基本块使用的 TCAM 资源；

H_i 表示 i 基本块使用的 HASH 资源；

A_i 表示 i 基本块使用的 ALU 资源；

Q_i 表示 i 基本块使用的 QUALIFY 资源；

Z 表示所有基本块所占用的最大流水线级数；

$$Z = max\{P_i | i \in [0,606]\} \tag{5-5}$$

N_k 表示偶数级流水线是否使用了 TCAM 资源：

$$N_k = \begin{cases} 1, \sum_{i \in A_k} T_i > 0 \\ 0, \sum_{i \in A_k} T_i = 0 \end{cases}, k \in [0, Z]\text{且为偶数} \quad (5\text{-}6)$$

结合上述符号，按照问题一的要求，需要使得所有基本块占用的流水线级数尽可能短，可以得出如下目标函数：

$$min\ Z = max\{P_i | i \in [0,606]\} \quad (5\text{-}7)$$

本题的目标是找到 607 个基本块的最优排布方案，使得占用的流水线级数最短，而 Z 为所有基本块中占用流水线级数的最大值，该目标函数是去寻找该值的最小值，符合本题的要求。

（2）约束条件

根据题目要求，模型需要满足以下约束条件：

数据依赖约束：在程序运行时，各个基本块之间可能需要读写同一个变量，而这就会导致基本块之间形成数据依赖，存在数据依赖的基本块在流水线上的顺序有一定的要求。具体而言，当基本块 i 和 j 存在写后读数据依赖或写后写数据依赖时，基本块 i 排布的流水线级数需要小于基本块 j 排布的级数，可得如下约束

$$P_i * D_1(i,j) < P_j, \quad i,j \in [0,606] \quad (5\text{-}8)$$

当基本块 i 和 j 存在读后写数据依赖时，基本块 i 排布的流水线级数需要小于或等于基本块 j 排布的级数，可得如下约束

$$P_i * D_2(i,j) \leq P_j, \quad i,j \in [0,606] \quad (5\text{-}9)$$

当两个基本块都需要读取一个变量时，不视为存在数据依赖关系。

控制依赖约束：在程序运行中，部分基本块的执行受到其他基本块的影响，对于这类基本块，其流水线级数也会受到约束。如果基本块 i 与基本块 j 存在控制依赖，则 i 排布的流水线级数需要小于或等于 j 排布的流水线级数，因此可得如下约束

$$P_i * C(i,j) \leq P_j, \quad i,j \in [0,606] \quad (5\text{-}10)$$

每一级流水线的资源约束条件：

$$\sum_{i \in A_k} T_i \leq 1, \ k \in [0, Z] \quad (5\text{-}11)$$

$$\sum_{i \in A_k} H_i \leq 2, \ k \in [0, Z] \quad (5\text{-}12)$$

$$\sum_{i \in A_k} A_i \leq 56, \ k \in [0, Z] \quad (5\text{-}13)$$

$$\sum_{i \in A_k} Q_i \leq 64, \ k \in [0, Z] \quad (5\text{-}14)$$

折叠级数资源约束：按照题目要求，流水线第 0 级与第 16 级，第 1 级与第 17 级，...，第 15 级与第 31 级为折叠级数，折叠的两级 TCAM 资源加起来最大为 1，HASH 资源加起来最大为 3，于是可得如下约束

$$\sum_{i \in A_k} T_i + \sum_{i \in A_{k+16}} T_i \leq 1, \ k \in [0,16) \quad (5\text{-}15)$$

$$\sum_{i \in A_k} H_i + \sum_{i \in A_{k+16}} H_i \leq 3, \ k \in [0,16) \quad (5\text{-}16)$$

偶数级数资源约束：使用 TCAM 资源的偶数流水线级数量不超过 5，可得如下约束

$$\sum_{k=0}^{Z} N_k \leq 5，k\text{为偶数} \quad (5\text{-}17)$$

在本模型中，基本块编号、流水线级数均为非负整数。

综上所述，针对问题一可以建立如下的芯片资源排布优化模型：

$$min\ Z = max\{P_i | i \in [0,606]\}$$

s.t.

$$P_i * D_1(i,j) < P_j ,\ \forall i,j \in [0,606]$$
$$P_i * D_2(i,j) \leq P_j ,\ \forall i,j \in [0,606]$$
$$P_i * C(i,j) \leq P_j ,\ \forall i,j \in [0,606]$$

$$\sum_{i \in A_k} T_i \leq 1，\forall k \in [0,Z]$$

$$\sum_{i \in A_k} H_i \leq 2，\forall k \in [0,Z]$$

$$\sum_{i \in A_k} A_i \leq 56，\forall k \in [0,Z]$$

$$\sum_{i \in A_k} Q_i \leq 64，\forall k \in [0,Z]$$

$$\sum_{i \in A_k} T_i + \sum_{i \in A_{k+16}} T_i \leq 1，\forall k \in [0,16)$$

$$\sum_{i \in A_k} H_i + \sum_{i \in A_{k+16}} H_i \leq 3，\forall k \in [0,16)$$

$$\sum_{k=0}^{Z} N_k \leq 5，k\text{为偶数}$$

5.1.3 模型求解

对于芯片资源排布模型的求解，考虑到该模型复杂度高、计算量大、约束条件多，使用求解器求解无法满足问题需求，所以本文首先尝试采用遗传算法进行求解。

遗传算法是一种现代优化算法，其原理是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法。该算法不需要目标函数满足可导或连续等限定条件，适用范围广，寻优能力好，已经广泛应用于各大领域中。

在遗传算法的优化求解过程中，本文将 607 个基本块的流水线级数对应为算法的基因位点，将每一个基本块先随机放置在一个流水线级上，然后依次判断是否满足资源约束条件、数据依赖约束条件、控制依赖约束条件，将满足所有约束条件的个体放入初始种群开始迭代优化。在迭代过程中，我们发现迭代优化的效率很低，并且收敛性差、收敛速度慢。分析算法发现原因有以下两点：

（1）在初始个体随机生成的情况下，能够满足所有约束条件的个体数量较少，导致初始种群的生成较为困难，需要改进算法的种群生成方式。

（2）每一次迭代都需要判断约束条件是否满足，而每一个基本块的判断都会耗费大量资源，这导致了迭代优化的效率很低。并且在判断控制依赖和数据依赖约束时，需要反复进行判断是否满足了约束条件。例如，在判断控制依赖约束时，基本块 *i* 与基本块 *j* 有控制依赖，则 *i* 的流水线级数需要小于等于 *j* 的流水线级数，而 *j* 又有可能与其他基本块有控制依赖关系，这就使得需要不断更新 *i* 的流水线级数约束，这大大增加了判

断约束条件的复杂性，导致了迭代优化的效率很差且收敛性差。

基于上述存在的问题，我们提出了结合贪婪算法的改进遗传算法来进行求解。贪婪算法是指在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，算法得到的是在某种意义上的局部最优解。通过引入贪婪算法，可以大幅降低算法的复杂度。

在改进算法中，会首先从 0 级流水线开始寻找目前的最佳选择，即只考虑 0 级，使得 0 级的资源利用率最高并且排布尽可能多的基本块。算法首先通过遗传算法的方式寻找 0 级流水线所有满足约束条件需要同时排布在一级的基本块，将这些需要同时排布在一级的基本块划分为一组。例如假定 *i* 基本块排布在 0 级，则所有流水线级数需要小于等于 *i* 的基本块也都需要排布在 0 级。

在进行约束判断时，改进算法通过将资源约束和控制依赖、数据依赖结合起来的方式简化了约束的判断。基于贪婪算法的思想，将不满足资源约束条件的基本块组直接剔除，这可以大幅降低约束判断的计算量，算法程序运行显示计算每一级的基本块编号平均只需要数十秒。

改进算法设置适应度函数为各种资源占用量与排布的基本块数量的加权和，从而得到能够最大化 0 级的资源利用率和排布基本块数量的基本块编号，将这些基本块去除之后再从 1 级流水线寻找目前的最优选择，不断重复直至排布完所有的基本块。

改进算法的求解步骤如下所示：

Step1. 初始化种群，调整种群个体基因生成 1 的概率为基因长度的倒数，这样可以使得基本块的排布尽可能靠前，而不是纯随机生成，便于遗传算法的迭代优化，提高迭代效率；

Step2. 将资源约束和控制依赖、数据依赖结合起来判断是否满足约束条件，输出满足约束条件且需要同时排布在一级的基本块组；

Step3. 求出适应度值最大的基本块组；

Step4. 去除已经排布好的基本块，若还有未排布的基本块则跳转至 *Step1* 计算下一级流水线的排布。

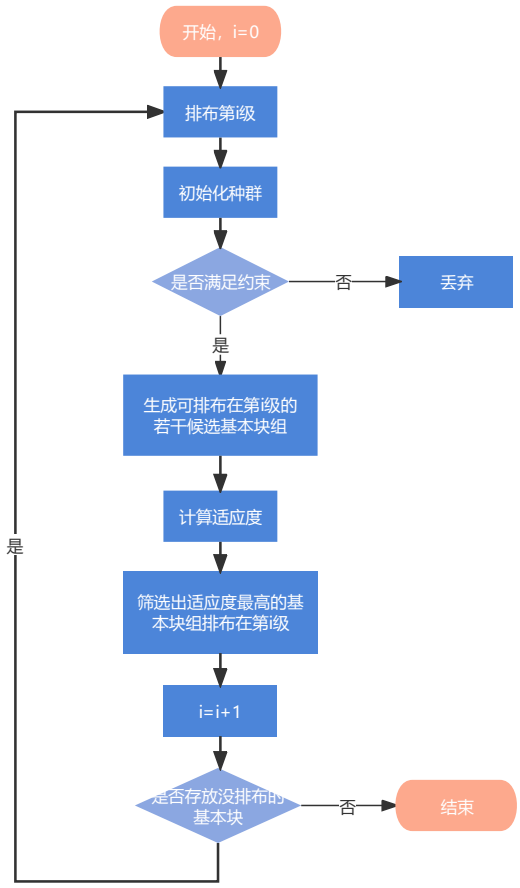

图 5-1 算法流程图

改进算法的流程图如图 5-1 所示。改进算法基于贪婪思想进行改进，大幅降低了计算复杂度，并且可以得到一个满意解。

5.1.4 求解结果与分析

（1）基本块排布情况

采用改进算法进行求解，最后得到问题一的所有基本块在满足约束条件的前提下需要占用 0 级到 39 级共 40 级流水线，每一级流水线的排布情况如图 5-1 所示。

从图中可以看出，由于采用的是基于贪婪思想的改进遗传算法，所以级数靠前的流水线会被安排更多的符合约束条件的基本块，从而得到级数靠前的流水线最高的适应度得分。对于级数靠后的流水线来说，它们被分配到的基本块数量并不多，可以看到后 10 级流水线大部分一级只排布了一个基本块，这些基本块更多地是因为约束关系而必须排布到新一级的流水线上。

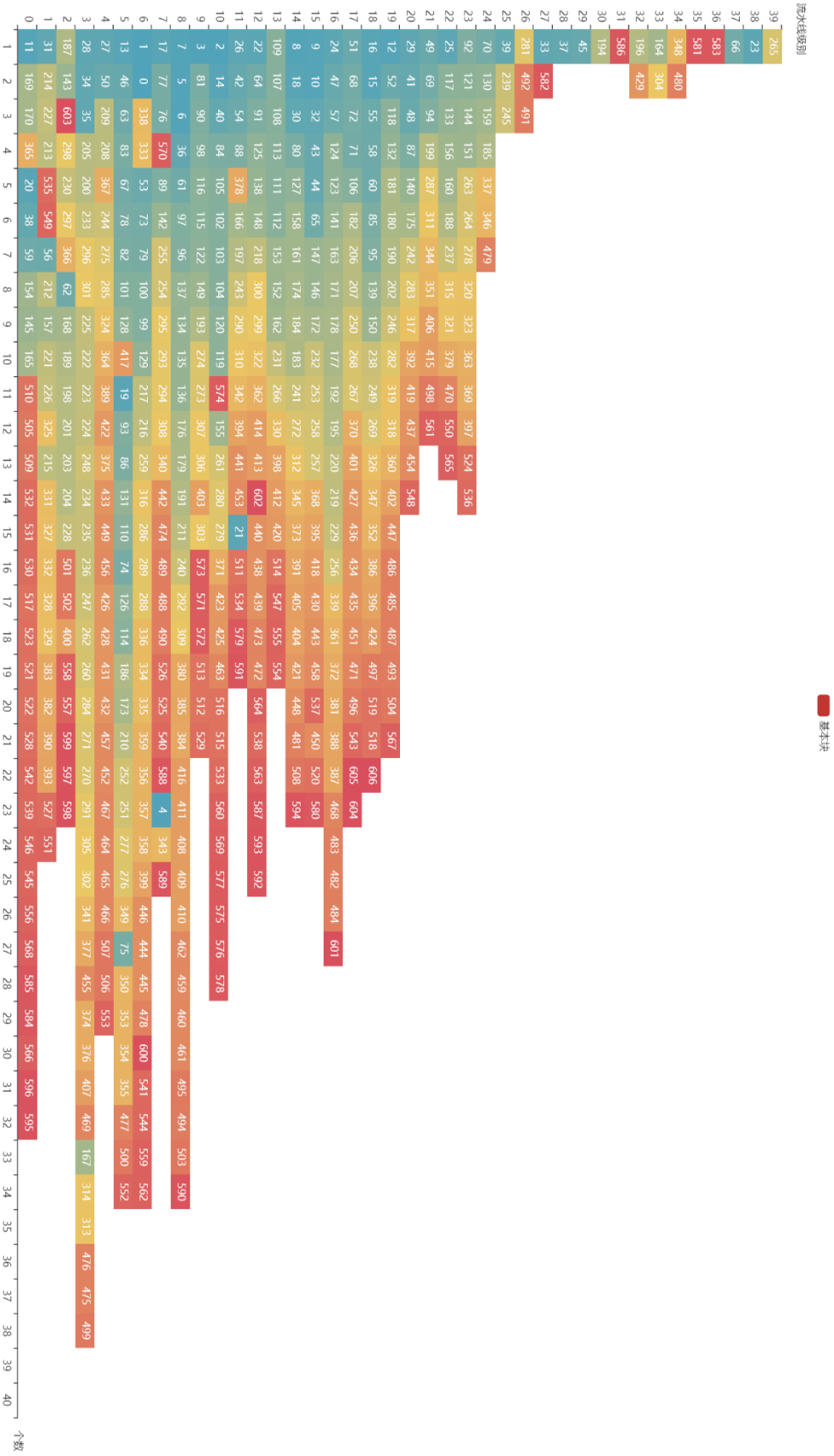

图 5-1 基本块排布情况

（2）各流水线级资源利用情况

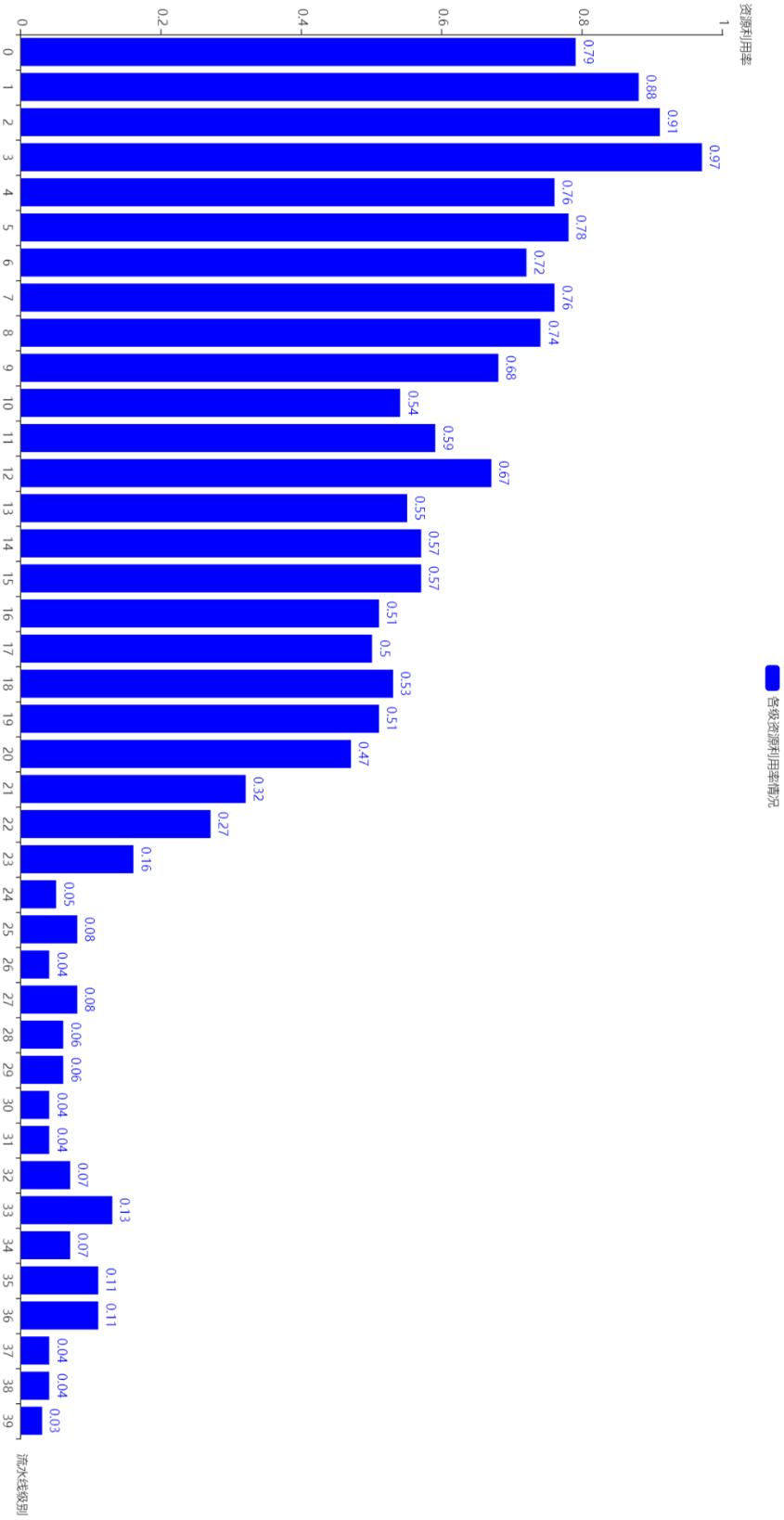

图 5-2 各流水线级资源利用率

上图展示了各流水线级的资源利用率，从图中可以看出级数靠前的流水线资源利用率明显高于级数靠后的流水线资源利用率，这主要是由于适应度函数设置为四种资源的资源利用率和基本块数的加权和，基于贪婪思想的改进算法会从第 0 级开始选择能使当前级适应度函数值最大的基本块组。而第 1 级至第 3 级资源利用率比第 0 级高是因为有很多基本块不能放在第 0 级，这些基本块需要其他的基本块级数小于它的级数，而这将无法做到，所以会被排除掉，当第 0 级放入一些基本块后，很多模块的限制就会解除，资源利用率就会变高一些。

从图中还可以观察到级数在 16 之后的流水线资源利用率会出现一次下降，并在 32 级之后又有一次小幅上升，这主要是由于折叠级数间的约束限制了基本块的排布，在 32 级之后去除掉折叠级数约束资源利用率上升符合题目预期。

（3）整个流水线的资源利用率情况

由于算法基于贪婪思想，每个流水线级的资源利用率呈现出由高到低的趋势，图 5-3 计算了整个流水线的总体资源占用率，其占用率为 39.5%。这主要是由于贪婪算法会将基本块尽可能的塞满级数靠前的流水线，导致级数靠后的流水线资源利用率普遍偏低。

图 5-3 整体资源利用率

本题最后得出的占用级数为 40 级，而从每个基本块所需的 HASH 资源来看，607 个基本块总共需要 63 的 HASH 资源。而对于问题一来说，考虑到折叠级数资源约束，仅分配 HASH 资源就需要 40 级流水线，也就是说最优解的流水线级数也应大于等于 40。因此改进算法得出的排布方式是能使占用级数最短的一种排布方式。

5.2 问题二的建模与求解

5.2.1 问题二模型的建立

问题二与问题一相比需要多考虑两个基本块是否在同一条执行流程上。如果由一个

基本块出发可以到达另一个基本块则两基本块在一条执行流程上，反之不在一条执行流程上。对于这种不在一条执行流程上的基本块，可以共享 HASH 资源和 ALU 资源。据此，需要修改问题一的部分约束条件，重新建立模型。

（1）目标函数

问题二相比问题一没有增加新的优化目标，仍以流水线占用级数最短为优化目标：

$$min\ Z = max\{P_i | i \in [0,606]\} \quad (5\text{-}18)$$

（2）确定约束条件

在给出约束条件前首先给定如下符号定义：

L_k 表示在 k 级流水线上所有的执行流程；

l 表示一个集合，其中存储了所有在一条执行流程上的基本块编号。

由题意可得，本题的数据依赖和控制依赖约束条件与问题一相同，不作更改，主要改变流水线的资源约束条件。

对于资源约束条件，不在一条执行流程上的基本块可以共享 HASH 资源和 ALU 资源，而对于 TCAM 资源与 QUALIFY 资源的约束条件也同问题一。于是根据问题二约束条件 2、3 可得：

$$\max_{l \in L_k} \sum_{i \in l} H_i \ \leq 2\ ,\ \ k \in [0, Z] \quad (5\text{-}19)$$

$$\max_{l \in L_k} \sum_{i \in l} A_i \ \leq 56\ ,\ \ k \in [0, Z] \quad (5\text{-}20)$$

对于折叠级数的 HASH 资源约束也需要进行修改：

$$\max_{l \in L_k} \sum_{i \in l} H_i \ + \max_{l \in L_{k+16}} \sum_{i \in l} H_i \ \leq 3\ ,\ \ k \in [0,16) \quad (5\text{-}21)$$

对于偶数级数资源约束不需要修改，最后可以建立如下模型：

$$min\ Z = max\{P_i | i \in [0,606]\}$$

s.t.

$$P_i * D_1(i,j) < P_j\ ,\quad \forall\ i,j \in [0,606]$$
$$P_i * D_2(i,j) \leq P_j\ ,\quad \forall\ i,j \in [0,606]$$
$$P_i * C(i,j) \leq P_j\ ,\quad \forall\ i,j \in [0,606]$$

$$\sum_{i \in A_k} T_i \ \leq 1\ ,\ \ \forall k \in [0, Z]$$

$$\max_{l \in L_k} \sum_{i \in l} H_i \ \leq 2\ ,\ \ k \in [0, Z]$$

$$\max_{l \in L_k} \sum_{i \in l} A_i \ \leq 56\ ,\ \ k \in [0, Z]$$

$$\sum_{i \in A_k} Q_i \ \leq 64\ ,\ \ \forall k \in [0, Z]$$

$$\sum_{i \in A_k} T_i + \sum_{i \in A_{k+16}} T_i \leq 1\ ,\ \ \forall k \in [0,16)$$

$$\max_{l \in L_k} \sum_{i \in l} H_i \ + \max_{l \in L_{k+16}} \sum_{i \in l} H_i \ \leq 3\ ,\ \ k \in [0,16)$$

$$\sum_{k=0}^{Z} N_k \ \leq 5\ ,\ \ k\text{为偶数}$$

5.2.2 模型求解

由于问题二的模型与问题一的模型类似，所以问题二的模型求解我们仍然采用基于贪婪思想的改进遗传算法进行求解，求解思路与问题一一致。

考虑问题二的假设修改对 HASH 和 ALU 资源进行了限制，因此我们只需对在不满足 HASH 和 ALU 资源限制的种群进行进一步求解。当 HASH 和 ALU 资源不满足限制时候，对该种群个体进行路径搜索，之后对重复路径进行筛选，可以得出不重复路径。

对那些不重复路径求解 HASH 和 ALU 资源，找出最大值与限制进行比较，如果满足限制则通过。

问题二的求解步骤如下：

Step1. 初始化种群，调整种群个体基因生成 1 的概率为基因长度的倒数，这样可以使得基本块的排布尽可能靠前，而不是纯随机生成，便于遗传算法的迭代优化，提高迭代效率；

*Step2.*判断控制依赖、数据依赖是否满足约束条件，对在同一条执行流程上的基本块判断资源约束条件，输出满足约束条件且需要同时排布在一级的基本块组；

Step3. 求出适应度值最大的基本块组；

Step4. 去除已经排布好的基本块，若还有未排布的基本块则跳转至 *Step1* 计算下一级流水线的排布。

5.2.3 结果分析

（1）基本块排布情况

采用改进算法进行求解，最后得到问题二的所有基本块在满足修改后约束条件的前提下需要占用共 23 级流水线，每一级流水线的排布情况如图 5-4 所示。

从图中可以看出，由于采用的是基于贪婪思想的改进遗传算法，所以与问题一相似级数靠前的流水线会被安排更多的符合约束条件的基本块，从而得到级数靠前的流水线最高的适应度得分。对于级数靠后的流水线来说，它们被分配到的基本块数量会相对较少，但相比问题一得益于不同执行流程可以共享资源，所以可以多分配一些基本块。

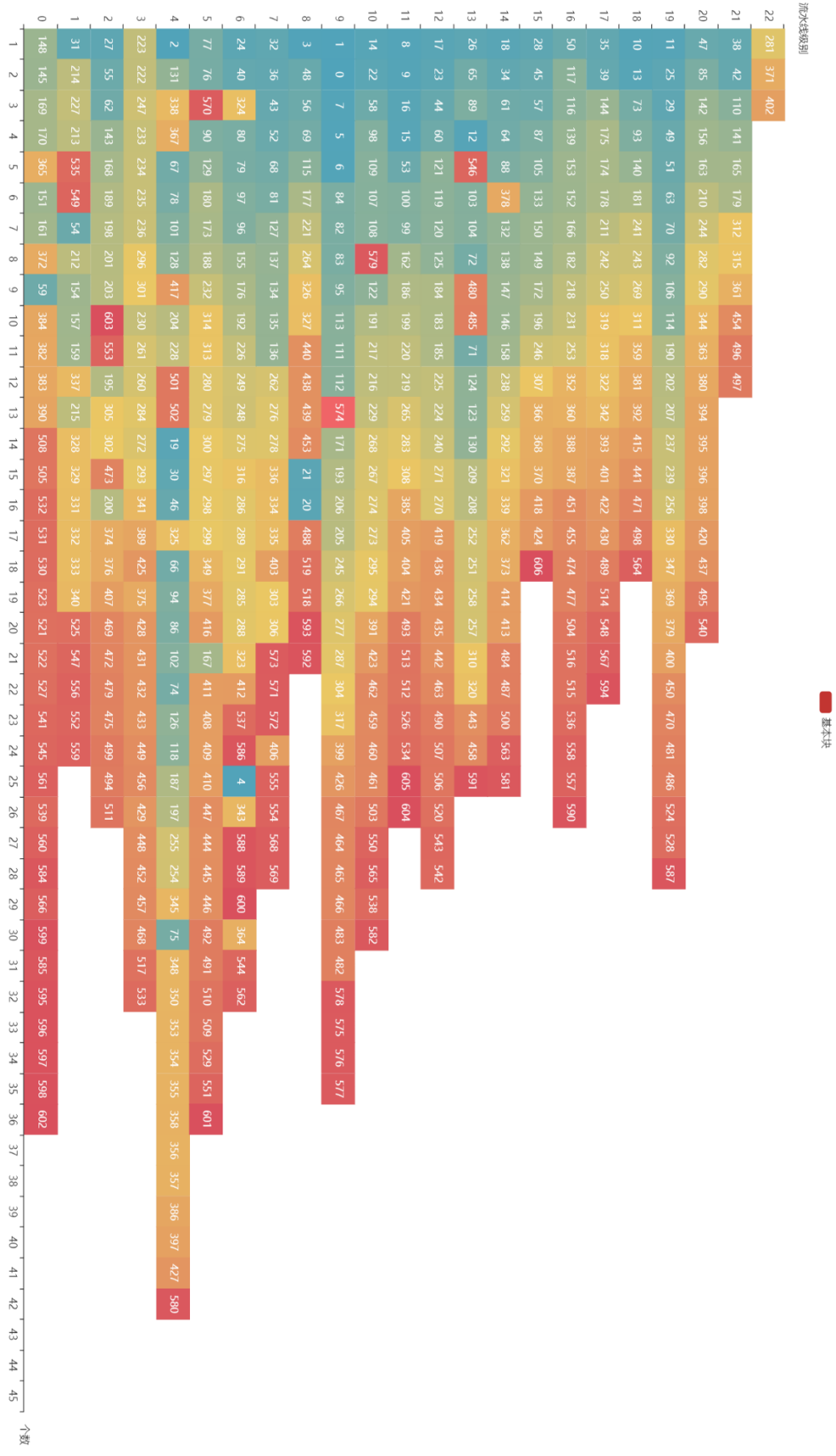

图 5-4 基本块排布情况

（2）各流水线级资源利用情况

上图展示了各流水线级的资源利用率，从图中可以看出级数靠前的流水线资源利用率相对高于级数靠后的流水线资源利用率，这与问题一是相似的。与问题一相比，问题二的流水线各级资源利用率分布更加均匀。

从图中还可以观察到级数在 16 之后的流水线资源利用率会与问题一类似出现一次下降，这主要是由于折叠级数间的约束限制了基本块的排布，符合题目预期。

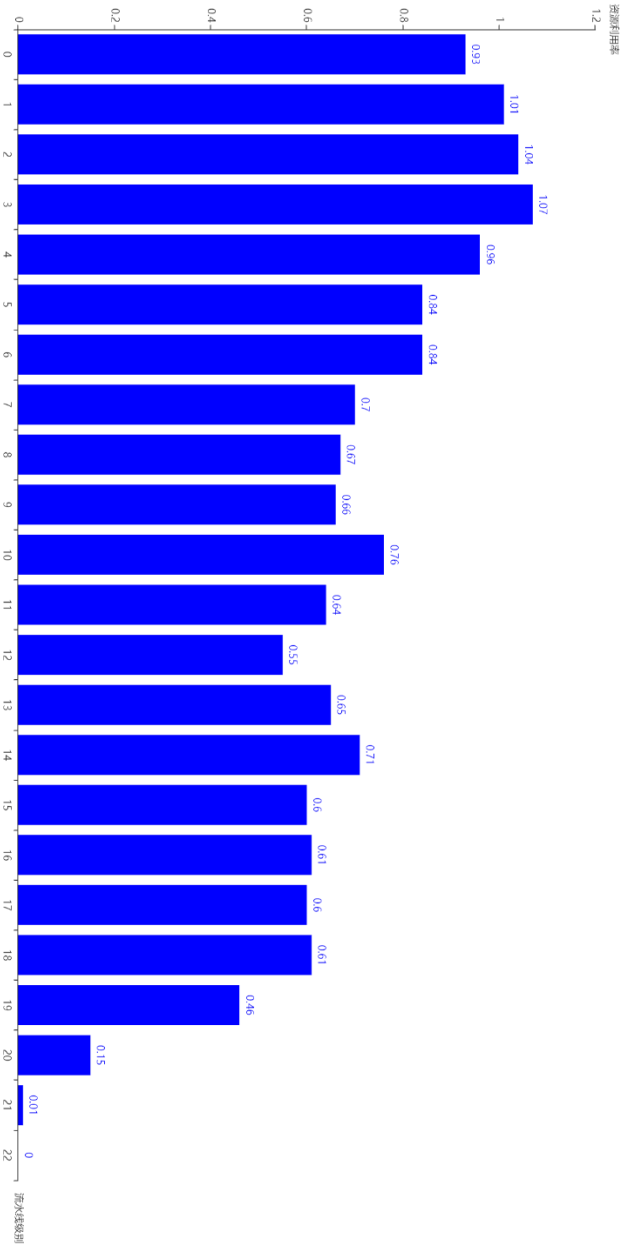

图 5-5 各流水线级资源利用情况

（3）整个流水线的资源利用率情况

图 5-6 展示了整个流水线的资源利用率，其值为 65.52%。可以发现流水线整体的资源利用率相比问题一提升了许多，这主要是由于在不同执行流程可以共享资源。

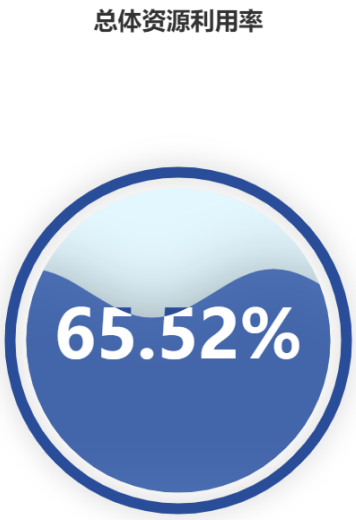

图 5-6 总体资源占用率

六、总结

本题以使占用的流水线级数最短为优化目标，针对问题一和问题二分别建立了两个多约束组合优化模型。

1、问题一的模型是以流水线级数最短为优化目标，在满足控制依赖、数据依赖和资源约束的条件进行基本块的排布，建立 0-1 整数规划模型，送入遗传算法进行求解。在求解过程中，遇到了约束条件判断复杂、收敛性差的问题，于是提出了基于贪婪思想的改进遗传算法，通过改进算法大大降低计算的复杂度，在较短的时间内给出了模型的局部最优解。

2、问题二在问题一的基础上更改了部分约束条件，对于资源约束条件需要进行重新规划。本文对 HASH 和 ALU 资源不满足限制的情况下，对该种群个体进行路径搜索，之后对重复路径进行筛选，可以得出不重复路径。通过对不重复路径求解 HASH 和 ALU 资源，找出最大值与限制进行比较，求得满足约束条件的基本块。求解思路与问题一相一致。

本文采取的基于贪婪思想的改进遗传算法模型加快了模型的求解，最后得出问题一的解为 40 级数，问题二的解为 23 级数，总体的基本块排布从前往后呈现逐级递减的态势，符合贪婪算法的思想。另外求解结果也给出了每一级的 4 种资源占用情况并计算每一级的资源利用率，各级资源利用率的累加和除以总级数得到总体资源利用率分别为 39.5%、65.52%，总体的排布结果满足题目要求。

七、参考文献

1. Bosshart P, Daly D, Gibb G, et al. P4: Programming protocol-independent packet processors[J]. ACM SIGCOMM Computer Communication Review, 2014, 44(3): 87-95.
2. Bosshart P, Gibb G, Kim H S, et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN[J]. ACM SIGCOMM Computer Communication Review, 2013, 43(4): 99-110.
3. 姚辉萍;赵雷;李莹;杨季文.一种改进的计算控制依赖的算法[J].计算机应用与软件,2010,v.27,19-21+40.
4. 刘力贞. 遗传算法在分布式约束优化问题中的应用研究[D].重庆大学,2020.
5. 杨子兰,朱娟萍,李睿.资源受限最小赋权树形图的一种贪婪分解启发式算法[J].西南师范大学学报(自然科学版),2017,42(08):18-24.
6. 丁可,徐言民,关宏旭,李诗杰,李柏苇.基于贪心-遗传优化算法的中长期船舶进出港调度优化[J/OL].武汉理工大学学报(交通科学与工程版):1-10[2022-10-09].

八、附录

附录 1：问题的求解

问题一主程序：

```
import numpy as np
import csv
import pandas as pd

CROSSOVER_RATE = 0.6
MUTATION_RATE = 0.01
N_GENERATIONS = 100
control1 = np.load('control.npy')
data_ziyuan = []
df = pd.read_csv('data_depend3.csv', index_col=0)
with open('data1.csv', encoding='utf-8-sig') as f:
    for row in csv.reader(f, skipinitialspace=True):
        data_ziyuan.append(row)

def ziyuan1(block):                    #输入快的序号，输出相应的资源数
    a = data_ziyuan[block+1]
    return a[1:5]


def SSS(num, temp):                        #查询数据依赖
    list_shujuyilai = []
    for i in range(607):
        Shuju = True
        if temp[i] == 0:
            if df.iloc[i, num] == 1:
                list_shujuyilai.append(i)
            if df.iloc[i, num] == -1:
                Shuju = False
    return Shuju, list_shujuyilai

def judge(block1, block2):
    """
    判定两个 block 是否在同一执行流程上
    :param block1:
    :param block2:
    :return:bool
    """
    df = pd.read_csv('先后顺序.csv', index_col = 0).astype('int')
    if df.iloc[block1, block2] == 1:
```

```
            flag1 = True    #在
        else:
            flag1 = False
        if df.iloc[block2, block1] == 1:
            flag2 = True    #不在
        else:
            flag2 = False
        return flag1|flag2

def Kongzhi(temp):                      #返回控制依赖的初始表
        list_K1 = []
        for i in range(607):
            list_K2 = []
            list_K2.append(i)
            if temp[i] == 0:
                for j in range(607):
                    if temp[j] == 0:
                        if control1[j, i] == 1:
                            list_K2.append(j)
                list_K1.append(list_K2)
        return list_K1
def Jiaocha_ziyuan(list1, list2):           #根据 list1  查询 list2，对 list1 进行添加
        A1 = 0
        for i in range(len(list1)):     #对 list1 进行遍历
            for j in list1[i]:              #list1[i]中的元素查询
                for h in range(len(list2)):
                    if j == list2[h][0]:
                        for m in list2[h]:
                            if m not in list1[i]:
                                A1 = A1 + 1
                                list1[i].append(m)
                        break
        return list1, A1

def Ziyuan_xianzhi(list1):
        list_ziyuanxianzhi = []
        for i in range(len(list1)):
                Ziyuanxianzhi1 = 0
                ziyuan2 = np.zeros(4)
                for j in list1[i]:
                    temp1 = np.array(ziyuan1(j))
                    for t1 in range(4):
                        ziyuan2[t1] = ziyuan2[t1] + int(temp1[t1])
                if ziyuan2[0] > list_zongziyuan1[0]:
```

```
                Ziyuanxianzhi1 = 1
            if ziyuan2[1] > list_zongziyuan1[1]:
                Ziyuanxianzhi1 = 1
            if ziyuan2[2] > list_zongziyuan1[2]:
                Ziyuanxianzhi1 = 1
            if ziyuan2[3] > list_zongziyuan1[3]:
                Ziyuanxianzhi1 = 1
            if Ziyuanxianzhi1 == 0:
                list_ziyuanxianzhi.append(list1[i])
    return list_ziyuanxianzhi
def Kongzhi_list(list1):
    for i in range(606):
        u = 0
        list1, u = Jiaocha_ziyuan(list1, list1)
        list1 = Ziyuan_xianzhi(list1)
        print('控制 u', u)
        if u == 0:
            break
    return list1


def Shuju(temp):
    list_shuju1 = []
    for i in range(607):
        if temp[i] == 0:
            list_shuju2 = []
            list_shuju2.append(i)
            A3, list_shuju3 = SSS(i, temp)
            if A3 and temp[i] == 0:
                for j in list_shuju3:
                    if temp[j] == 0:
                        list_shuju2.append(j)
                list_shuju1.append(list_shuju2)
    return list_shuju1

def guolv(list):
    # 还需要加一步，去掉不能放在第一级的
    list_guolv = []
    for i in range(len(list)):
        list_guolv.append(list[i][0])
    list_guolv1 = []
    for i in range(len(list)):
        s = 0
        for j in list[i]:
```

```
                if j not in list_guolv:
                    s = 1
            if s == 0:
                list_guolv1.append(list[i])
    return list_guolv1
def Shuju_list(list):
    for i in range(607):
        list,u = Jiaocha_ziyuan(list, list)
        list = guolv(list)
        print('资源 u', u)
        list = Ziyuan_xianzhi(list)
        if u == 0:
            break
    return list

def Kongzhi_Shuju(list_K,list_S):
    for i in range(607):
        list_zong, u0 = Jiaocha_ziyuan(list_K, list_S)
        list_zong, u1 = Jiaocha_ziyuan(list_zong, list1)
        list_zong = Ziyuan_xianzhi(list_zong)
        if (u0+u1) == 0:
            break
    return list_zong

def crossover_and_mutation(pop, CROSSOVER_RATE=0.8):
    new_pop = []
    for father in pop:  # 遍历种群中的每一个个体，将该个体作为父亲
        child = father  # 孩子先得到父亲的全部基因（这里我把一串二进制串的那些0，1 称为基因）
        if np.random.rand() < CROSSOVER_RATE:  # 产生子代时不是必然发生交叉，而是以一定的概率发生交叉
            mother = pop[np.random.randint(POP_SIZE)]  # 再种群中选择另一个个体，并将该个体作为母亲
            cross_points = np.random.randint(low=0, high=DNA_SIZE)  # 随机产生交叉的点
            child[cross_points:] = mother[cross_points:]  # 孩子得到位于交叉点后的母亲的基因
        mutation(child)  # 每个后代有一定的机率发生变异
        new_pop.append(child)

    return new_pop

def mutation(child, MUTATION_RATE=0.003):
    if np.random.rand() < MUTATION_RATE:  # 以 MUTATION_RATE 的概率进行变异
```

```
        mutate_point = np.random.randint(0, DNA_SIZE)  # 随机产生一个实数，代表要变异基因的位置
        child[mutate_point] = child[mutate_point] ^ 1  # 将变异点的二进制为反转

def F(pop,list):
    list_F = []
    for i in range(POP_SIZE):
        list_F1 = []
        for j in range(len(list)):
            if pop[i, j] == 1:
                for t in list[j]:
                    if t not in list_F1:
                        list_F1.append(t)
        list_F.append(list_F1)
    ziyuan_F = np.zeros(POP_SIZE)
    for i in range(POP_SIZE):
        ziyuan_F1 = np.zeros(4)
        for j in list_F[i]:
            temp1 = np.array(ziyuan1(j))
            for t1 in range(4):
                ziyuan_F1[t1] = ziyuan_F1[t1] + int(temp1[t1])
        if ziyuan_F1[0] > list_zongziyuan1[0]:
            ziyuan_F[i] = 0
            continue
        if ziyuan_F1[1] > list_zongziyuan1[1]:
            ziyuan_F[i] = 0
            continue
        if ziyuan_F1[2] > list_zongziyuan1[2]:
            ziyuan_F[i] = 0
            continue
        if ziyuan_F1[3] > list_zongziyuan1[3]:
            ziyuan_F[i] = 0
            continue
        ziyuan_F[i] = ziyuan_F1[0]*64 + ziyuan_F1[1]*32 + ziyuan_F1[2]*1.1143 + ziyuan_F1[3] + len(list_F[i])*7
    return ziyuan_F, list_F
def get_fitness(pop, list):
    pred, list_F = F(pop,list)
    return pred, list_F
    # return pred - np.min(pred)+1e-3  # 求最大值时的适应度
    # return np.max(pred) - pred + 1e-3  # 求最小值时的适应度，通过这一步 fitness 的范围为[0, np.max(pred)-np.min(pred)]
```

```
def select(pop, fitness):    # nature selection wrt pop's fitness
    # pop_size 种群数量
    if fitness.sum() != 0:
        idx = np.random.choice(np.arange(POP_SIZE), size=POP_SIZE, replace=True,
                               p=(fitness) / (fitness.sum()))
    else:
        return pop

    return pop[idx]


def GA(list):
    fitness_GA = []
    list_GA = []
    pop = np.random.randint(DNA_SIZE, size=(POP_SIZE, DNA_SIZE))
    for i in range(POP_SIZE):
        for j in range(DNA_SIZE):
            if pop[i,j] == (DNA_SIZE-1):
                pop[i,j] == 1
            else:
                pop[i,j] == 0
    for _ in range(N_GENERATIONS):   # 迭代 N 代
        pop = np.array(crossover_and_mutation(pop, CROSSOVER_RATE))
        fitness,list_GA_jieduan = get_fitness(pop, list)
        max_fitness_index = np.argmax(fitness)
        fitness_GA.append(fitness[max_fitness_index])
        if list_GA_jieduan[max_fitness_index] != 0:
            list_GA.append(list_GA_jieduan[max_fitness_index])
        else:
            list_GA.append([])

        pop = select(pop, fitness)   # 选择生成新的种群

        print("max_fitness:", fitness[max_fitness_index])
    return fitness_GA, list_GA
def jisuanziyuan(list):
    ziyuan_F1 = np.zeros(4)
    for i in list:
        temp1 = np.array(ziyuan1(i))
        for t1 in range(4):
            ziyuan_F1[t1] = ziyuan_F1[t1] + int(temp1[t1])
    return ziyuan_F1
list_jishuziyuan = []
N_oushuziyuan = 0
```

```
temp = np.zeros(607)
list_block = []
list_zongziyuan = []
for i in range(100):
    list_zongziyuan1 = [1, 2, 56, 64]
    if i >= 16 and i <= 31:
        list_zongziyuan1[0] = 1 - list_jishuziyuan[i-16][0]
        list_zongziyuan1[1] = 3 - list_jishuziyuan[i - 16][1]
    if N_oushuziyuan > 4:
        if i % 2 == 0:
            list_zongziyuan1[0] = 0
    list1 = Kongzhi(temp)
    list1 = Kongzhi_list(list1)
    list2 = Shuju(temp)
    list2 = Shuju_list(list2)
    list_zong = Kongzhi_Shuju(list1, list2)
    POP_SIZE = len(list_zong)*10
    DNA_SIZE = len(list_zong)
    print('输入遗传算法的样本长度', DNA_SIZE)
    A11, B11 = GA(list_zong)
    list_block.append(B11[np.argmax(A11)])
    for j in list_block[i]:
        temp[j] = temp[j]+1
    list_jishuziyuan.append(jisuanziyuan(list_block[i]))
    if i % 2 == 0:
        N_oushuziyuan += list_jishuziyuan[i][0]
        print('偶数级数的 0 号资源占用情况', N_oushuziyuan)
    print('第',i,'级','放置子程序块数目', len(list_block[i]))
    if len(list_block[i]) == 0:
        break
```

问题二主程序：

```
import numpy as np
import csv
import pandas as pd

CROSSOVER_RATE = 0.6
MUTATION_RATE = 0.01
N_GENERATIONS = 10
control1 = np.load('control.npy')
data_ziyuan = []
df = pd.read_csv('data_depend3.csv', index_col=0)
with open('data1.csv', encoding='utf-8-sig') as f:
    for row in csv.reader(f, skipinitialspace=True):
```

```
        data_ziyuan.append(row)

def ziyuan1(block):                    #输入快的序号，输出相应的资源数
    a = data_ziyuan[block+1]
    return a[1:5]


df1 = pd.read_csv('先后顺序.csv', index_col = 0).astype('int')
def judge1(block1, block2):
    """
    判定两 2 是否在 1 后面
    :param block1:
    :param block2:
    :return:bool
    """
    if df1.iloc[block1, block2] == 1:
        flag1 = True    #在
    else:
        flag1 = False
    return flag1

class Solution:
    #graph 表示一张图
    #start 表示起始节点
    #target 表示终止节点
    #cut 表示最大路长
    def searchpaths(self, start, target, cut, graph):
        paths = []
        path = []
        # graph = {1:[2, 5], 2:[3, 4], 3:[4, 5], 5:[6]}
        # print(graph[1])
        # graph = {'A': ['B', 'C'], 'B': ['C'], 'C': ['D']}

        def dfs(start, target):
            #如果起始节点不在图中，则结束
            if start not in graph:
                return
            #如果路长大于阈值，则结束
            if len(path) >= cut:
                return
            #遍历起始节点的邻居节点
            for neighbor in graph[start]:
                #如果起始节点没有在单条 path 出现，则追加
                if neighbor not in path:
```

```
                path.append(neighbor)
            #出现环，则结束
            else:
                return
            #如果单条 path 没有出现在 paths 里且满足终点等于 target，则在结果
中追加
            if path not in paths and path[-1] == target:
                #拷贝
                paths.append(path[:])
            #以新的邻居节点 dfs
            dfs(neighbor, target)
            #last 的追加了 neighbor，需要返回
            path.pop()
    path.append(start)
    dfs(start, target)
    return paths

def find_path(list):
    temp_xunlu = np.zeros(607)
    Record1 = []
    for i in range(len(list)):
        Record2 = []
        Record2.append(list[i])
        for j in range(len(list)):
            if judge1(list[i], list[j]) and i != j:
                Record2.append(list[j])
        Record1.append(Record2)
    return Record1

def Zhuanhuan(list):
    dict1 = {}
    temp_shunxu = np.zeros(len(list))
    qidian1 = []
    zhongdian1 = []
    for j in range(len(list)):
        list_max = []
        for i in range(len(list)):
            if temp_shunxu[i] == 0:
                if len(list[i]) > len(list_max):
                    list_max = list[i]
        qidian = []
        zhongdian = []
        qidian.append(list_max[0])
        for i in range(len(list)):
```

```
                if list[i][0] in list_max:
                    temp_shunxu[i] = 1
                    if len(list[i]) == 1:
                        zhongdian.append(list[i][0])
            qidian1.append(qidian)
            zhongdian1.append(zhongdian)
            if 0 not in temp_shunxu:
                break




    for i in range(len(list)):
        if len(list[i]) == 1:
            continue
        else:
            value_dict = []
            for j in range(len(list[i])-1):
                value_dict.append(int(list[i][j+1]))
            dict1[list[i][0]] = value_dict
    return qidian1,zhongdian1,dict1


def ALL_path(qidian,zhongdian,list_xunxu,dict_allpath):
    ALL_rob_path = []
    for i in range(len(qidian)):
        for j in range(len(qidian[i])):
            for k in range(len(zhongdian[i])):
                for sr in list_xunxu:
                    if qidian[i][j] == sr[0]:
                        cut_zuizhong = len(sr)
                ALL_rob_path.append(Solution().searchpaths(start=qidian[i][j],
target=zhongdian[i][k], cut=cut_zuizhong,
                                                           graph=dict_allpath))
    return ALL_rob_path

def baohanyu(list1,list2):
    Flag1 = True
    for i in list1:
        if i not in list2:
            Flag1 = False
    return Flag1

def path_quchong(all_path):
```

```
    shengyu_path = []
    for i in range(len(all_path)):
        temp_quchu = np.zeros(len(all_path[i]))
        for j in range(len(all_path[i])):
            biaozhi_quchu = 0
            for t in range(len(all_path[i])):
                if temp_quchu[t] == 1:
                    continue
                if baohanyu(all_path[i][j], all_path[i][t]) and len(all_path[i][j]) < len(all_path[i][t]):
                    temp_quchu[j] = 1
                    biaozhi_quchu = 1
                    break
            if biaozhi_quchu == 0:
                shengyu_path.append(all_path[i][j])
    return shengyu_path




def  xunzhaoluxian(list):
    list_xunxu = find_path(list)
    qidian, zhongdian, dict_allpath = Zhuanhuan(list_xunxu)
    all_path = ALL_path(qidian,zhongdian,list_xunxu,dict_allpath)
    shengyu_path = path_quchong(all_path)
    return shengyu_path

def Ziyuan_liucheng(list):
    max_HASH = 0
    max_ALU = 0
    for i in range(len(list)):
        ziyuan_F1 = np.zeros(4)
        for j in list[i]:
            temp1 = np.array(ziyuan1(j))
            for t1 in range(4):
                ziyuan_F1[t1] = ziyuan_F1[t1] + int(temp1[t1])
        if ziyuan_F1[1] > max_HASH:
            max_HASH = ziyuan_F1[1]
        if ziyuan_F1[2] > max_ALU:
            max_ALU = ziyuan_F1[2]
    return max_HASH, max_ALU
```

```
def SSS(num, temp):                          #查询数据依赖
    list_shujuyilai = []
    for i in range(607):
        Shuju = True
        if temp[i] == 0:
            if df.iloc[i, num] == 1:
                list_shujuyilai.append(i)
            if df.iloc[i, num] == -1:
                Shuju = False
    return Shuju, list_shujuyilai

def judge(block1, block2):
    """
    判定两个 block 是否在同一执行流程上
    :param block1:
    :param block2:
    :return:bool
    """
    df = pd.read_csv('先后顺序.csv', index_col = 0).astype('int')
    if df.iloc[block1, block2] == 1:
        flag1 = True    #在
    else:
        flag1 = False
    if df.iloc[block2, block1] == 1:
        flag2 = True    #不在
    else:
        flag2 = False
    return flag1|flag2

def Kongzhi(temp):                           #返回控制依赖的初始表
    list_K1 = []
    for i in range(607):
        list_K2 = []
        list_K2.append(i)
        if temp[i] == 0:
            for j in range(607):
                if temp[j] == 0:
                    if control1[j, i] == 1:
                        list_K2.append(j)
```

```
                list_K1.append(list_K2)
    return list_K1
def Jiaocha_ziyuan(list1, list2):            #根据 list1 查询 list2，对 list1 进行添加
    A1 = 0
    for i in range(len(list1)):     #对 list1 进行遍历
        for j in list1[i]:              #list1[i]中的元素查询
            for h in range(len(list2)):
                if j == list2[h][0]:
                    for m in list2[h]:
                        if m not in list1[i]:
                            A1 = A1 + 1
                            list1[i].append(m)
                    break
    return list1, A1

def Ziyuan_xianzhi(list1):
    list_ziyuanxianzhi = []
    for i in range(len(list1)):
            Ziyuanxianzhi1 = 0
            ziyuan2 = np.zeros(4)
            for j in list1[i]:
                temp1 = np.array(ziyuan1(j))
                for t1 in range(4):
                    ziyuan2[t1] = ziyuan2[t1] + int(temp1[t1])
            if ziyuan2[0] > list_zongziyuan1[0]:
                Ziyuanxianzhi1 = 1
            if ziyuan2[1] > list_zongziyuan1[1]:
                Ziyuanxianzhi1 = 1
            if ziyuan2[2] > list_zongziyuan1[2]:
                Ziyuanxianzhi1 = 1
            if ziyuan2[3] > list_zongziyuan1[3]:
                Ziyuanxianzhi1 = 1
            if Ziyuanxianzhi1 == 0:
                list_ziyuanxianzhi.append(list1[i])
    return list_ziyuanxianzhi
def Kongzhi_list(list1):
    for i in range(606):
        u = 0
        list1, u = Jiaocha_ziyuan(list1, list1)
        list1 = Ziyuan_xianzhi(list1)
        print('控制 u', u)
        if u == 0:
            break
    return list1
```

```
def Shuju(temp):
    list_shuju1 = []
    for i in range(607):
        if temp[i] == 0:
            list_shuju2 = []
            list_shuju2.append(i)
            A3, list_shuju3 = SSS(i, temp)
            if A3 and temp[i] == 0:
                for j in list_shuju3:
                    if temp[j] == 0:
                        list_shuju2.append(j)
                list_shuju1.append(list_shuju2)
    return list_shuju1

def guolv(list):
    # 还需要加一步，去掉不能放在第一级的
    list_guolv = []
    for i in range(len(list)):
        list_guolv.append(list[i][0])
    list_guolv1 = []
    for i in range(len(list)):
        s = 0
        for j in list[i]:
            if j not in list_guolv:
                s = 1
        if s == 0:
            list_guolv1.append(list[i])
    return list_guolv1
def Shuju_list(list):
    for i in range(607):
        list, u = Jiaocha_ziyuan(list, list)
        list = guolv(list)
        print('资源 u', u)
        list = Ziyuan_xianzhi(list)
        if u == 0:
            break
    return list

def Kongzhi_Shuju(list_K,list_S):
    for i in range(607):
        list_zong, u0 = Jiaocha_ziyuan(list_K, list_S)
        list_zong, u1 = Jiaocha_ziyuan(list_zong, list1)
```

```
            list_zong = Ziyuan_xianzhi(list_zong)
            if (u0+u1) == 0:
                break
    return list_zong

def crossover_and_mutation(pop, CROSSOVER_RATE=0.8):
    new_pop = []
    for father in pop:  # 遍历种群中的每一个个体，将该个体作为父亲
        child = father  # 孩子先得到父亲的全部基因（这里我把一串二进制串的那些0，1 称为基因）
        if np.random.rand() < CROSSOVER_RATE:  # 产生子代时不是必然发生交叉，而是以一定的概率发生交叉
            mother = pop[np.random.randint(POP_SIZE)]  # 再种群中选择另一个个体，并将该个体作为母亲
            cross_points = np.random.randint(low=0, high=DNA_SIZE)  # 随机产生交叉的点
            child[cross_points:] = mother[cross_points:]  # 孩子得到位于交叉点后的母亲的基因
        mutation(child)  # 每个后代有一定的机率发生变异
        new_pop.append(child)

    return new_pop

def mutation(child, MUTATION_RATE=0.003):
    if np.random.rand() < MUTATION_RATE:  # 以 MUTATION_RATE 的概率进行变异
        mutate_point = np.random.randint(0, DNA_SIZE)  # 随机产生一个实数，代表要变异基因的位置
        child[mutate_point] = child[mutate_point] ^ 1  # 将变异点的二进制为反转

def F(pop,list):
    list_F = []
    for i in range(POP_SIZE):
        list_F1 = []
        for j in range(len(list)):
            if pop[i, j] == 1:
                for t in list[j]:
                    if t not in list_F1:
                        list_F1.append(t)
        list_F.append(list_F1)
    ziyuan_F = np.zeros(POP_SIZE)
    for i in range(POP_SIZE):
        ziyuan_F1 = np.zeros(4)
        for j in list_F[i]:
            temp1 = np.array(ziyuan1(j))
```

```
                for t1 in range(4):
                    ziyuan_F1[t1] = ziyuan_F1[t1] + int(temp1[t1])
            if ziyuan_F1[0] > list_zongziyuan1[0]:
                ziyuan_F[i] = 0
                continue
            if ziyuan_F1[3] > list_zongziyuan1[3]:
                ziyuan_F[i] = 0
                continue
            if ziyuan_F1[1] > list_zongziyuan1[1] or ziyuan_F1[2] > list_zongziyuan1[2]:
                path_shengyu = xunzhaoluxian(list_F[i])
                max_HASH, max_ALU = Ziyuan_liucheng(path_shengyu)
                if max_HASH > list_zongziyuan1[1] or max_ALU > list_zongziyuan1[2]:
                    ziyuan_F[i] = 0
                    continue
                else:
                    ziyuan_F[i] = ziyuan_F1[0] * 64 + ziyuan_F1[1] * 32 + ziyuan_F1[2] *
1.1143 + ziyuan_F1[3] + len(list_F[i]) * 7
                    continue
            ziyuan_F[i] = ziyuan_F1[0]*64 + ziyuan_F1[1]*32 + ziyuan_F1[2]*1.1143 +
ziyuan_F1[3] + len(list_F[i])*7
        return ziyuan_F, list_F
def get_fitness(pop, list):
        pred, list_F = F(pop,list)
        return pred, list_F
        # return pred - np.min(pred)+1e-3   # 求最大值时的适应度
        # return np.max(pred) - pred + 1e-3   # 求最小值时的适应度，通过这一步 fitness 的
范围为[0, np.max(pred)-np.min(pred)]


def select(pop, fitness):    # nature selection wrt pop's fitness
        # pop_size 种群数量
        if fitness.sum() != 0:
            idx = np.random.choice(np.arange(POP_SIZE), size=POP_SIZE, replace=True,
                                   p=(fitness) / (fitness.sum()))
        else:
            return pop

        return pop[idx]


def GA(list):
        fitness_GA = []
        list_GA = []
        pop = np.random.randint(DNA_SIZE, size=(POP_SIZE, DNA_SIZE))
```

```
    for i in range(POP_SIZE):
        for j in range(DNA_SIZE):
            if pop[i,j] == (DNA_SIZE-1):
                pop[i,j] == 1
            else:
                pop[i,j] == 0
    for ssr in range(N_GENERATIONS):  # 迭代 N 代
        pop = np.array(crossover_and_mutation(pop, CROSSOVER_RATE))
        fitness,list_GA_jieduan = get_fitness(pop, list)
        max_fitness_index = np.argmax(fitness)
        fitness_GA.append(fitness[max_fitness_index])
        if list_GA_jieduan[max_fitness_index] != 0:
            list_GA.append(list_GA_jieduan[max_fitness_index])
        else:
            list_GA.append([])

        pop = select(pop, fitness)  # 选择生成新的种群

        print("max_fitness:", fitness[max_fitness_index])
    return fitness_GA, list_GA
def jisuanziyuan(list):
    ziyuan_F1 = np.zeros(4)
    for i in list:
        temp1 = np.array(ziyuan1(i))
        for t1 in range(4):
            ziyuan_F1[t1] = ziyuan_F1[t1] + int(temp1[t1])
    return ziyuan_F1

list_jishuziyuan = []
N_oushuziyuan = 0
temp = np.zeros(607)
list_block = []
list_zongziyuan = []
for i in range(100):
    list_zongziyuan1 = [1, 2, 56, 64]
    if i >= 16 and i <= 31:
        list_zongziyuan1[0] = 1 - list_jishuziyuan[i-16][0]
        list_zongziyuan1[1] = 3 - list_jishuziyuan[i - 16][1]
    if N_oushuziyuan > 4:
        if i % 2 == 0:
            list_zongziyuan1[0] = 0
    list1 = Kongzhi(temp)
    list1 = Kongzhi_list(list1)
    list2 = Shuju(temp)
```

```
list2 = Shuju_list(list2)
list_zong = Kongzhi_Shuju(list1, list2)
POP_SIZE = 40
DNA_SIZE = len(list_zong)
print('输入遗传算法的样本长度', DNA_SIZE)
A11, B11 = GA(list_zong)
list_block.append(B11[np.argmax(A11)])
for j in list_block[i]:
    temp[j] = temp[j]+1
list_jishuziyuan.append(jisuanziyuan(list_block[i]))
if i % 2 == 0:
    N_oushuziyuan += list_jishuziyuan[i][0]
    print('偶数级数的 0 号资源占用情况', N_oushuziyuan)
print('第',i,'级','放置子程序块数目', len(list_block[i]))
if len(list_block[i]) == 0:
    break
```

附录 2：数据预处理

生成基本块先后顺序表：

```
import numpy as np
import pandas as pd
import math
# import sys
import csv
class Solution:
    #graph 表示一张图
    #start 表示起始节点
    #target 表示终止节点
    #cut 表示最大路长
    def searchpaths(self, start, graph):
        path = []

        def dfs(start):
            #如果起始节点不在图中，则结束
            if start not in graph:
                return
            #遍历起始节点的邻居节点
            for neighbor in graph[start]:
                #如果起始节点没有在单条 path 出现，则追加
                print(neighbor)
                if neighbor not in path:
                    path.append(neighbor)

                #以新的邻居节点 dfs
```

```
                    dfs(neighbor)
        dfs(start)
        return path
data = []                                   #导入有向无环数据
with open('data3.csv', encoding='utf-8-sig') as f:
    for row in csv.reader(f, skipinitialspace=True):
        data.append(row)
path = np.array(data)                       #将数据 DataFrame 转化为 numpy
record = np.zeros((607, 607))               #将路径保存到 numpy 中
a = []                                      #用于记录所有方向
for i in range (607):                       #将所有路径保存到 list 中
    for j in range(11):
        if str.isdigit(path[i, j+1]):
            a.append([i, int(path[i, j+1])])
        else:
            break
for i in a:
    record[i[0],i[1]] = 1
dict = {}
for i in range(607):
    y = []
    for o in range(607):
        if record[i,o] != 0:
            y.append(o)
    dict[i] = y
successively = np.zeros((607,607))
for i in range(607):
    print(i)
    temp = Solution().searchpaths(start=i, graph=dict)
    for j in temp:
        successively[i,j] = 1
a = np.zeros((607,607))
for i in range(607):
    for j in range(607):
        a[i,j] = successively[i,j]+successively[j,i]
import pandas as pd
data1 = pd.DataFrame(successively)
data1.to_csv('先后顺序.csv')
```

生成控制依赖表：

```
import numpy as np
import pandas as pd
import math
# import sys
```

```
import csv
class Solution:
    #graph 表示一张图
    #start 表示起始节点
    #target 表示终止节点
    #cut 表示最大路长
    def searchpaths(self, start, graph):
        path = []

        def dfs(start):
            #如果起始节点不在图中，则结束
            if start not in graph:
                return
            #遍历起始节点的邻居节点
            for neighbor in graph[start]:
                #如果起始节点没有在单条 path 出现，则追加
                print(neighbor)
                if neighbor not in path:
                    path.append(neighbor)

                #以新的邻居节点 dfs
                    dfs(neighbor)
        dfs(start)
        return path
data = []                                   #导入有向无环数据
with open('data3.csv', encoding='utf-8-sig') as f:
    for row in csv.reader(f, skipinitialspace=True):
        data.append(row)
path = np.array(data)                       #将数据 DataFrame 转化为 numpy
record = np.zeros((607, 607))               #将路径保存到 numpy 中
a = []                                      #用于记录所有方向
for i in range (607):                       #将所有路径保存到 list 中
    for j in range(11):
        if str.isdigit(path[i, j+1]):
            a.append([i, int(path[i, j+1])])
        else:
            break
for i in a:
    record[i[0],i[1]] = 1
dict = {}
for i in range(607):
    y = []
    for o in range(607):
        if record[i,o] != 0:
```

```
                y.append(o)
        dict[i] = y
successively = np.zeros((607,607))
for i in range(607):
        print(i)
        temp = Solution().searchpaths(start=i, graph=dict)
        for j in temp:
                successively[i,j] = 1
a = np.zeros((607,607))
for i in range(607):
        for j in range(607):
                a[i,j] = successively[i,j]+successively[j,i]
import pandas as pd
data1 = pd.DataFrame(successively)
data1.to_csv('先后顺序.csv')
control1 = np.zeros((607,607))
for t in range(607):
        y = []
        temp1 = np.zeros(607)
        for o in range(607):
                if record[t, o] != 0:
                        y.append(o)
        if len(y) > 1:
                for j in y:
                        temp1[j] = temp1[j] + 1
                        temp1 = temp1 + successively[j,:]
                for i in range(607):
                        if temp1[i] > 0 and temp1[i] < len(y):
                                temp1[i] = 1
                        else:
                                temp1[i] = 0
        control1[t,:] = temp1
data2 = pd.DataFrame(control1)
data2.to_csv('控制依赖.csv')
```

生成数据依赖表：

```
import pandas as pd
import numpy as np

def inter(list1, list2):
        """
        判断两个列表是否存在交集
        :param list1:
        :param list2:
```

```
    :return: bool
    """
    inter = set(list1)&set(list2)
    #True 代表存在交集，False 代表不存在
    if inter:
        flag = True
    else:
        flag = False
    return flag

def trans():
    df = pd.read_csv('out/data_depend2.csv', index_col = 0)
    for i in range(0, 607):
        col_index = get_columns(i)
        for j in col_index:
            if df.iloc[i, j] == "['D']":
                df.iloc[i, j] = 0    # 代表没有数据依赖
            elif df.iloc[i, j] == "['C']":
                df.iloc[i, j] = 1     # 存在依赖<=
            elif inter(df.iloc[i, j], ['A', 'B']) == True:
                #存在交集
                df.iloc[i, j] = -1     # 存在依赖<
            else:
                pass
    df.to_csv('out/data_depend3.csv')

def get_columns(row):
    """
    根据【先后顺序.csv】，输入行号，输出值为 1 的元素的列索引存入列表
row: 输入的行号
    :return: 存放列索引的列表
    """
    df = pd.read_csv('out/先后顺序.csv', index_col=0)
    lst = [item for item in df.iloc[row, :]]
    tmp = []
    tag = 0
    for i in lst:
        if i == 1:
            tmp.append(tag)
        else:
            pass
        tag += 1
    return tmp
```

```
def data_depend(tar):
    """
    判断两个 block 是否存在读写依赖
    :param tar:  接收的 block 对,元组类型
    :return:
    """
    df = pd.read_csv('data/attachment2.csv', header=None)
    a = tar[0]
    b = tar[1]

    #  存储每个 block 读写的变量
    list_a_w = (df.iloc[2*a, 1:]).dropna().to_list()[1:]
    list_a_r = (df.iloc[2*a+1, 1:]).dropna().to_list()[1:]
    list_b_w = (df.iloc[2*b, 1:]).dropna().to_list()[1:]
    list_b_r = (df.iloc[2*b+1, 1:]).dropna().to_list()[1:]

    list = [list_a_w, list_a_r, list_b_w, list_b_r]

    res = []       #存储判断结果
    flag = ['A', 'B', 'C', 'D']       #判断标志
    ww = inter(list_a_w, list_b_w)
    wr = inter(list_a_w, list_b_r)
    rw = inter(list_a_r, list_b_w)
    # rr = inter(list_a_r, list_b_r)

    if ww==True:
        res.append(flag[0])
    if wr==True:
        res.append(flag[1])
    if rw==True:
        res.append(flag[2])
    # if rr==True:
    #         # res.append(flag[3])
    #         pass

    # 以上条件都不成立时，表示不存在控制依赖，置为'D'
    if ww|wr|rw == True:
        pass
    else:
        res.append(flag[3])

    return res
```

```
#初始化表格
df = pd.DataFrame(index = [a for a in range(0, 607)], columns = [a for a in range(0, 607)])

for i in range(0, 607):
	col_index = get_columns(i)
	for j in col_index:
		res = data_depend(tar=(i,j))
		df.iloc[i, j] = res

df.to_csv('out/data_depend2.csv')
```