

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	贵州大学
参赛队号	22106570188
队员姓名	1.宋　耀
	2.张松鸿
	3.李晓庆

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研究

摘 要：

当前日益复杂国际形式下，芯片成了各个大国的必争高科技技术。PISA 作为目前主流的可交换变成芯片架构之一，在实际 PISA 架构设计时，为减少连线复杂度，往往对流水线各级资源以及各级流水线资源有多种多样约束。然而，芯片的各类资源均有限，研究高资源利用率的资源排布算法对编译器设计尤为重要。针对此问题本文从资源排布的 PISA 架构资源约束、流图中基本块约束出发，建立**单目标整数规划模型**，用于求解满足架构各流水级资源约束，任务流图中各基本块约束的同时提升 PISA 架构芯片资源排布中的资源利用率，**模型有效降低资源排布时复杂度**。

针对问题一，分析流图基本块的控制依赖约束、数据依赖约束、资源依赖约束，求解基本块间的**控制依赖矩阵、数据依赖约束矩阵**。引入**基本块流水级矩阵**作为决策变量，**最小化流水线级数**作为目标函数，将架构中各级流水线资源限制、各级流水线间资源限制、基本块间控制依赖、基本块间数据依赖作为约束条件，构建**单目标优化模型**。采用**基于规则的启发式算法**，依据架构资源优先的启发信息对模型进行求解。利用**分治思想与归并排序**算法，引入**基本块近似单调队列**，降低模型求解复杂度。模型最终求解得到**最小流水级数为 48。**

针对问题二，分析问题二与问题一的约束条件可知**问题一的解空间是问题二的解空间的子空间**。因此**将问题一求解出的最优解作为问题二的初始解**。为此，建立以基本块决策变量，最小化流水线级数为目标函数，每级流水线资源约束与**每级流水线上同一执行流程资源约束、折叠级流水线同一执行流程资源约束**、基本块间控制依赖、基本块间数据依赖

等约束作为约束条件，构建优化模型。采用基于启发式算法，依据架构资源优先的启发信息求解模型。利用深度优先搜索算法思想设计**流水线执行流程查找算法**实现求解每级流水线中的所有执行流程，并求解每级流水线中所有执行流程的 HASH、ALU 资源之和最大值。有效地降低了模型求解的时间复杂度。模型最终求解得到**最小流水线级数为 32。**

关键词：资源排布；单目标规划；基于规则的启发式算法；近似单调队列；分治思想；

目录

一、问题重述

1.1 问题背景

随着科技时代的发展，手机、计算机、机器人、电动汽车等一系列电子产品，都离不开芯片的支撑。在目前全球经济数字化、科技已成为各国国际战略博弈的重要战场，芯片产业并非仅仅是一个经济问题或者市场问题，成为国家之间用来经济制衡的武器。传统的交换芯片功能固定，随着网络协议的不断更新，传统交换芯片便不能满足新的需求，重新设计芯片到成产使用又会耗费相当大的时间和人力，为了解决此问题，诞生了可编程的交换芯片 PISA（Protocol Independent Switch Architecture）。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]。

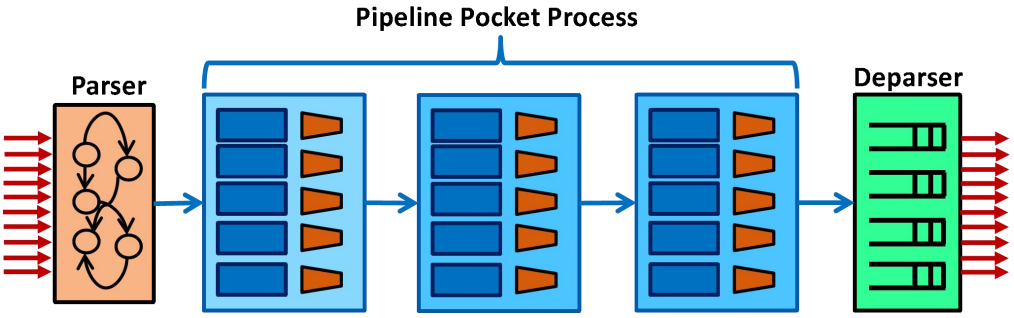

图 1　PISA 架构图

PISA 架构如图 1 所示，其包括报文解析（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。报文解析用于识别报文种类；多级的报文处理流水线用于修改报文数据，在实际的 PISA 架构芯片中，不同的芯片流水线的级数可能不同；报文重组用于报文重新组装。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。

1.2 待求解问题

在给出了各基本块在 P4 程序流程图中的邻接关系，各基本块占用的四类资源的数量，以及各基本块读写的变量信息，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，芯片的资源约束、资源排布时不能违反芯片的资源限制，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，并充分考虑各子问题的优化目标，以求**最大化芯片资源利用率**。

问题 1：给定资源约束条件如下：

（1）流水线每级的 TCAM 资源最大为 1；

（2）流水线每级的 HASH 资源最大为 2；

（3）流水线每级的 ALU 资源最大为 56；

（4）流水线每级的 QUALIFY 资源最大为 64；

（5）约定流水线第 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。注：如果需要的流水线级数超过 32 级，则从第 32 开始的级数不考虑折叠资源限制；

（6）有 TCAM 资源的偶数级数量不超过 5；

（7）每个基本块只能排布到一级。

在上述资源约束条件下进行资源排布，并以占用的**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 2：考虑如图 2 所示的流图，基本块 2 和基本块 3 不在一条执行流程上（因为基本块 1 执行完后要么执行基本块 2，要么执行基本块 3，基本块 2 和基本块 3 不可能都执行）。准确来说，在 P4 程序流程图中，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，基本块 2 和 3 中任意一个的 HASH 资源与 ALU 资源均不超过每级资源限制，基本块 2 和 3 即可排布到同一级。据此，对问题 1 中的约束条件（2）、（3）、（5）作如下更改：

（2）流水线每级中同一条执行流程上的基本块的 HASH 资源之和最大为 2；

（3）流水线每级中同一条执行流程上的基本块的 ALU 资源之和最大为 56；

（5）折叠的两级，对于 TCAM 资源约束不变，对于 HASH 资源，每级分别计算同一条执行流程上的基本块占用的 HASH 资源，再将两级的计算结果相加，结果不超过 3。

其它约束条件同问题 1，更改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。

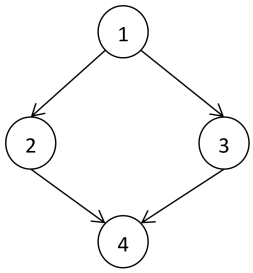

图 2　流图示例

二、问题分析

2.1 问题一分析

针对问题一，问题要求给出资源排布算法，期望使得在基本块的放置满足控制依赖约束、数据依赖约束、资源约束条件下，占用流水线级数尽可能短。据此建立单目标线性规划模型，模型的优化目标等价于资源利用率最大化，优化目标明确。

问题一的单目标优化模型思路分析如图 3 所示。为简化模型表述与建立，建立基本块在流水线级中的存在性变量矩阵、基本块之间的控制依赖变量矩阵、基本块之间的数据依赖变量矩阵。同时建立资源约束集进行建模，考虑使用基于规则的启发式算法求解模型。

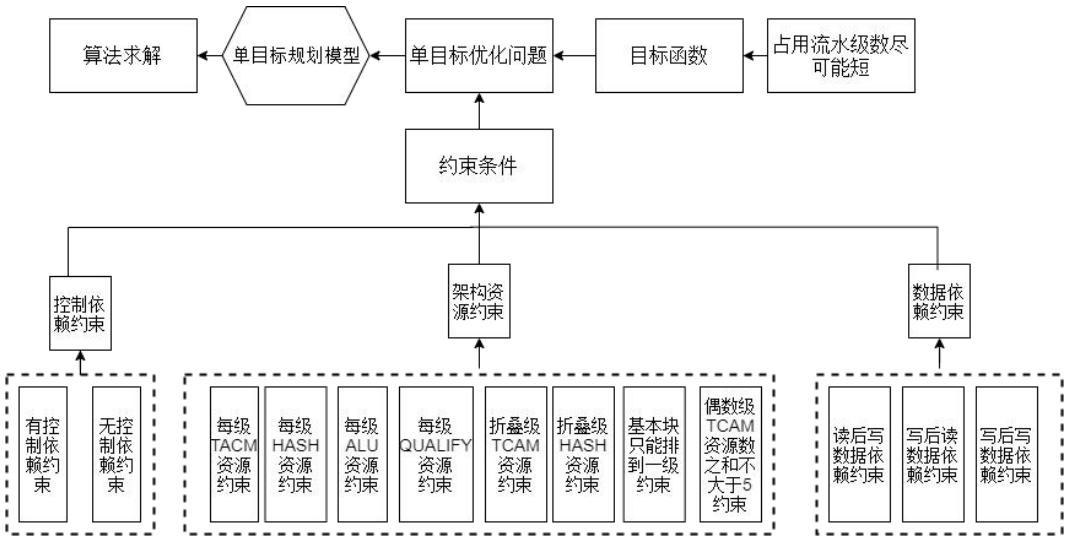

图 3 问题一分析思路分析图

2.2 问题二分析

针对问题二，在问题一的基础上改变资源约束集中的部分约束条件，保持控制依赖约束与数据依赖约束不动。为考虑每级流水线中同一执行流程变量的资源限制，引入流水线执行流程字典变量并加入模型约束条件。

问题二的思路分析如图 4 所示。问题二更改资源约束条件与问题一相比松弛了，使得问题二约束集构成的约束空间相较于问题一增大，流水线级数可进一步变短。将问题的排布结果作为问题二的初始排布方案，建立单目标规划模型，考虑设计基于规则启发式算法求解模型。

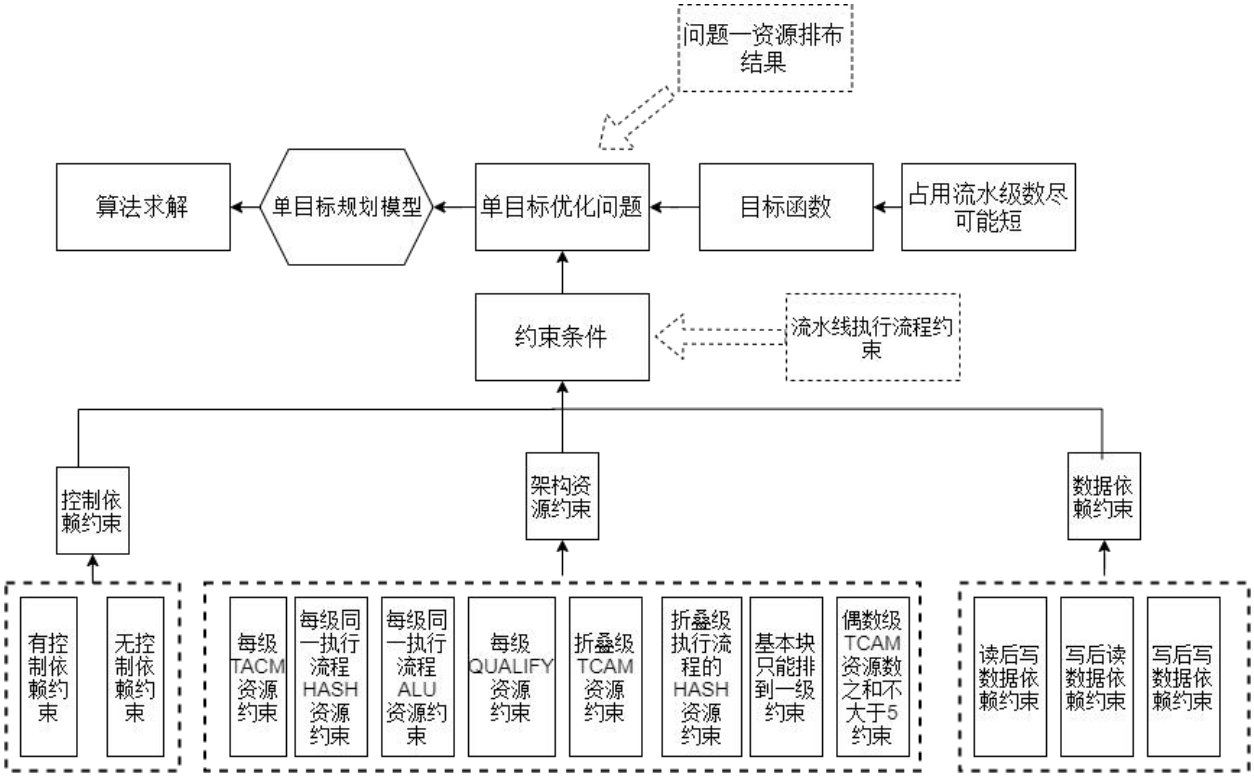

图 4 问题二思路分析图

三、模型假设

1、任意给定一个基本块都能在满足资源限制的情况下放置到某个适当的流水线；
2、假设不存在因控制依赖与数据依赖关系造成两个基本块所在流水线单元级数都互相小于对方的矛盾情况；
3、由基本块构建的流程图中不存在环，即假设芯片架构中任务流图为有向无环图；

四、符号说明

符号	符号含义	备注
i	第 i 个基本块，i 的顺序按照所给顺序集进行排序	
j	第 j 级流水线，j 的顺序按照流水线级数进行排序	
n	流图中基本块的总数	$n=607$
A_{ij}	第个 i 基本块是否在第 j 级流水线	
A_{pq}	第个 p 基本块是否在第 q 级流水线，其中 $p \in \{0,1,2,...,i-1,i+1,\cdots,n-1\}, 1 \le q \le j$	
C	基本块间控制依赖矩阵	
S	基本块间数据依赖矩阵	
P	基本块间路径存在性矩阵	
CT_i	第 i 个基本块的 TACM 资源需求量	
CH_i	第 i 个基本块的 HASH 资源需求量	
CA_i	第 i 个基本块的 ALU 资源需求量	
CQ_i	第 i 个基本块的 QUALIFY 资源需求量	
Z	优化目标函数	
Q_1	流图中所有基本块的四种资源消耗之和	
L	资源排布流水线级数	
Q_2	L 级流水的四种资源限制之和	
R	资源利用率	

注：表中未说明的符号变量以首次出现为准。

五、数据预处理

5.1 控制依赖预处理

依据各基本块在流图中的邻接基本块数表 attachment3.csv 数据，构建有向无环图。并利用有向无环图构建基本块控制依赖矩阵 C。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。”结合有向无环图，判断基本块间控制依赖存在性。如图 5 所示流图中，存在以下三种情形（为便于表述，以下使用节点表示基本块）：

(1) **两节点中，其中一节点的所有子节点都可经有向路径到达到另一节点，不存在控制依赖关系。**如图 5（a）中，B1 的子节点只有一个 B2,从子节点 B2 出发，都能分别找到一条路径到达点 B3、B4、B5、B6、B7,从而 B1 与 B3、B4、B5、B6、B7 均不存在控制依赖关系。B2 两个子节点 B3、B4,分别从 B2 的子节点 B3、B4 出发,能找到路径 B3-B5-B7,B4-B5-B7,从 B2 的每个子节点出发都能找到路径到达目标点 B7,从而 B2 与 B7 无控制依赖关系；

(2) **两节点间无任何有向路径相连，不存在控制依赖关系。**如图 5（b）中，B5 与 B6 无任何有向路径相连，从而 B5 与 B6 之间不存在控制依赖关系；

(3) **两节点中，其中一节点的至少一子节点经有向路径无法到达另一节点，存在控制依赖关系。**如图 5（b）中，B2 的子节点 B4 可以达到 B5，但 B2 子节点 B3 无法到达 B5，因此 B2 与 B5 间存在控制依赖关系。

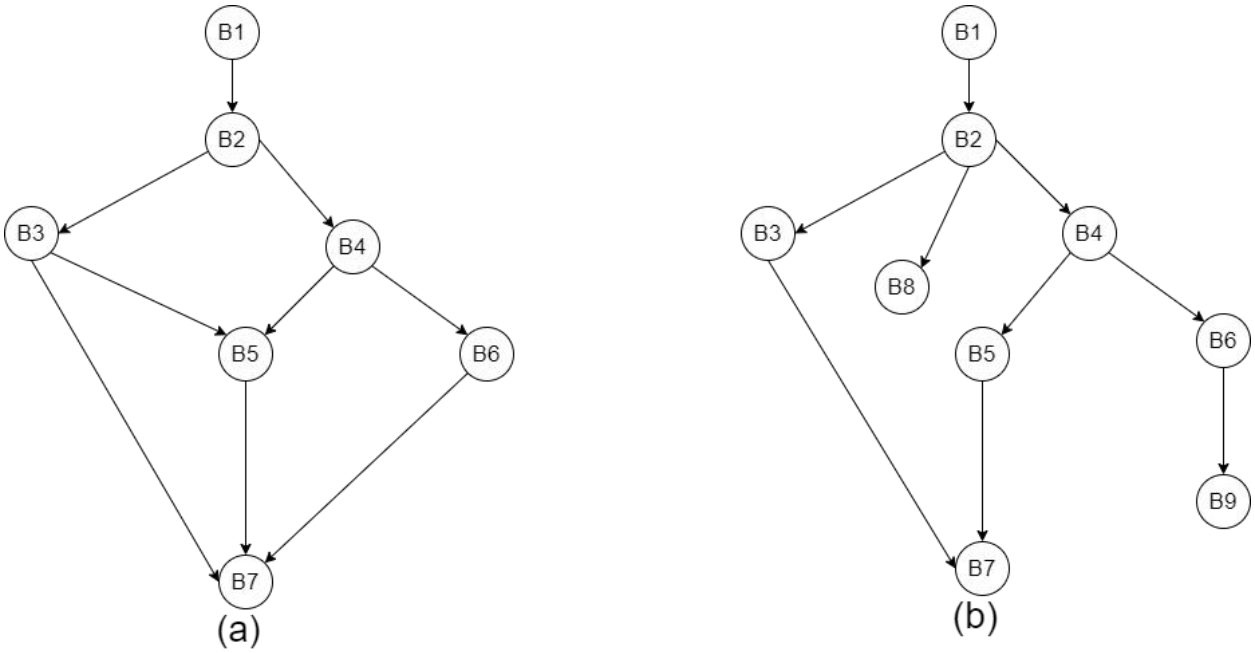

图 5 控制依赖示意流图

基于上述三种情形，利用**回溯思想，**判断任意给定的两节点之间是否存在控制依赖关系。以图 5（b）中判断起始节点 B2 与目标节点 B7 是否存在控制依赖为例，可由如下步骤求解得到：

Step1: 找出 B2 的所有子节点{B3, B4, B8}；

Step2: 任选 B2 的一个子节点 B3 为起点,按有向无环图的连接关系向下查找，有 B3-B7 路径存在,以 B3 为根节点的子树存在一条到 B7 的路劲，回退到 B3;

Step3: 从 B2 的剩余子节点中任选节点 B4,B4 有两个子节点{B5, B6},任选节点 B6 继续向下查找，到达叶子节点 B9 仍未探查到目标 B7,终止向下查找；回退至上一次选择的位置 B4，以 B5 为起点，向下查找到达 B7,以 B4 为根节点的子树存在路径 B4-B5-B7,回退到 B4;

Step4: 从 B2 的剩余子节点中选择节点 B8，向下查找到到达叶子节点 B9 仍未探查到目标 B7,终止向下查找；以 B8 为根节点的子树不能找到到达目标点 B7 的路径。终止整个判断过程，判定起始节点 B2 与目标节点 B7 存在控制依赖；

总结上述两个节点间的控制依赖关系判断流程，得到伪代码如表 1 所示。

表 1 判断两节点间控制依赖关系的伪代码

回溯法判断两点间控制依赖关系
Input:起点 A,终点 B，根据所有基本块的邻接关系生成邻接链表
Output: Flag,若 Flag=True, A 与 B 之间存在依赖关系；否则 A 与 B 不存在控制依赖
01.childM 存放 A 的所有子节点
02. For pop in childM:
03. stack 记录起点 A
04. visited 记录已经访问过的点 A
05. while len(stack)>0:
06. 从 stack 末尾中取出一个节点 ver
07. 如果取出的节点 ver 刚好是 B,则退出 while 循环
08. 用 nodes 记录节点 ver 的所有子节点
09. for node in nodes:
10. if node 刚好是 B,则跳出 for 循环
11. if node 从未访问过，则往 stack,visited 的末尾中加入 node
12. if 前面都没有遇到 B,则跳出 while 循环
13. if 前面的 while 循环中没有探查到节点 B
14. if A 与 B 之间不存在一条通路
15. 则判定 A 与 B 之间不存在控制依赖
16. else A 与 B 之间存在控制依赖
17.if 经由 childM 中每个节点都可找到通往目标点 B 的路径
18. 判定 A 与 B 之间不存在控制依赖

依据伪代码，设计算法求解控制依赖矩阵 C, C_{ij} 表示第 i 个基本块与第 j 个基本块之间的控制依赖关系的存在性，即有：

$$C_{ij}=\begin{cases}1, \text{基本块}i\text{与基本块}j\text{存在控制依赖关系}\\ 0, \text{基本块}i\text{与基本块}j\text{不存在控制依赖关系}\end{cases} \tag{1}$$

其中，$C_{ij}=0$ 表示第 i 号基本块与第 j 号基本块之间存在控制依赖关系，此时 i, j 所在流水线级数没有大小关系要求；$C_{ij}=1$ 则表示第 i 号基本块与第 j 号基本块之间存在控制依赖关

系，基本块i所在流水线单元级数小于等于基本块j所在流水线单元的级数；求解基本块间的控制依赖矩阵同时，保存任意两节点之间路径存在性矩阵,矩阵中的值含义如下式所示：

$$P_{ij}=\begin{cases}1,\text{基本块}i\text{与基本块}j\text{存在通路}\\0,\text{基本块}i\text{与基本块}j\text{不存在通路}\end{cases} \tag{2}$$

5.2 数据依赖预处理

由 5.1 节控制依赖预处理构建的有向无环图与基本块间路劲存在性矩阵，结合各基本块资源使用情况表 attachment2.csv，构建基本块之间的数据依赖矩阵S。两个基本块之间的数据依赖存在 8 种情形：

（1）两个基本块之间不存在数据依赖关系；
（2）两个基本块之间存在读后写数据依赖关系；
（3）两个基本块之间存在写后读数据依赖关系；
（4）两个基本块之间存在写后写数据依赖关系；
（5）两个基本块之间存在读后写与写后读数据依赖关系；
（6）两个基本块之间存在读后写与写后写数据依赖关系；
（7）两个基本块之间存在写后写与写后读数据依赖关系；
（8）两个基本块之间存在读后写、写后读、写后写数据依赖关系。

依据这 8 种数据依赖关系情况，设计求解基本块间数据依赖关系判断流程图如图 6 所示。

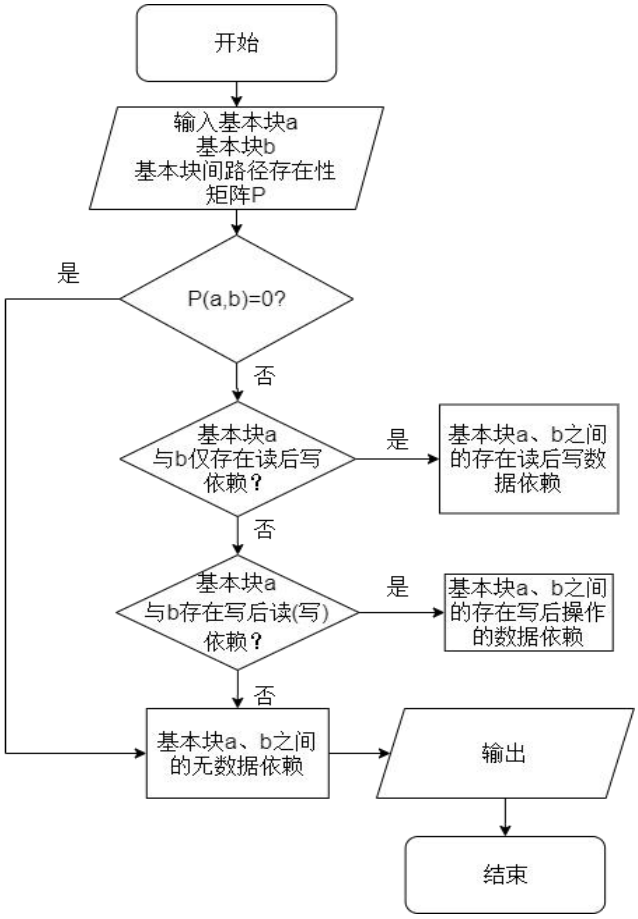

图 6 数据依赖判断流程图

依据图 6，求解得到基本块间的数据依赖矩阵 S，S 中的值 S_{ij} 表示第 i 个基本块与第 j 个基本块之间的数据依赖关系。取值如下式所示：

$$S_{ij}=\begin{cases}0\\1\\2\end{cases} \tag{3}$$

其中，$S_{ij}=0$ 代表基本块 i 与基本块 j 之间不存在数据依赖关系。$S_{ij}=1$ 代表基本块 i 与基本块 j 之间**仅存在读后写数据**依赖。$S_{ij}=2$ 代表基本块 i 与基本块 j 之间存在**写后读**或**写后写**数据依赖。

六、模型的建立与求解

6.1 问题一模型建立与求解

6.1.1 PISA 芯片资源排布模型建立

(一) 决策变量

A_{ij}：表示基本块 i 是否排布在流水线级 j 上，A_{ij} 为一个 $0-1$ 变量，其取值代表着实际的基本块排布方案。若第 i 个基本块放置到第 j 个流水线则对应 A_{ij} 取值为 1，第 i 个基本块并没有放置到第 j 个流水线上则取值为 0。由于不能将同一基本块排布到不同的流水线中，因此 A_{ij} 满足约束：

$$\sum_{i=0}^{n-1}A_{ij}=1,(j=0,1,2,\cdots,n-1) \tag{4}$$

(二) 目标函数

为了使基本块放置排布方案的流水线级数尽可能短，目标函数需要满足以下两个条件：

（1） 目标函数与决策变量 A_{ij} 之间存在联系；

（2） 目标函数与流水线级数 L 有关；

依据上述两个条件，目标函数设置为：

$$Z=\min L=\max\frac{Q_1}{Q_2}=\max\frac{Q_1}{123*f(\sum_{i=0}^{n-1}A_{i1},\sum_{i=0}^{n-1}A_{i1},...,\sum_{i=0}^{n-1}A_{in})} \tag{5}$$

其中 123 表示每级流水线的四种资源的限制量之和，Q_1 表示流图中所有基本块的四种资源

消耗量之和。函数 $f(x_1,x_2,...,x_n)$ 定义如式 6：

$$f(x_0,x_1,...,x_{n-1})=\sum_{i=0}^{n-1}f(\mathrm{x}_i)$$
$$f(x)=\begin{cases}1,x>0\\0,x=0\end{cases} \tag{6}$$

(三)约束条件

（1）TCAM、HASH、ALU、QUALIFY 四种资源约束：

根据已知要求,设第 i 个基本块上 TCAM、HASH、ALU、QUALIFY 四种资源上的资源量分别为 $CT_i,CH_i,CA_i,C\mathrm{Q}_i$，结合四种资源在流水线每级上的限制要求，则该四种资源在第 j 个流水单元上的资源限制约束可以表述为：

$$\sum_{i=0}^{n-1}C\mathrm{T}_i a_{ij}\le 1 \tag{7}$$

$$\sum_{i=0}^{n-1}CH_i a_{ij}\le 2 \tag{8}$$

$$\sum_{i=n}^{n-1}CA_i a_{ij}\le 56 \tag{9}$$

$$\sum_{i=0}^{n-1}CQ_i a_{ij}\le 64 \tag{10}$$

（2）流水线折叠层的资源约束：

约定流水线第 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，且流水线级数超过 32 级时，从第 32 级开始的级数都不考虑折叠资源对基本块分发对方的限制作用。具有折叠关系的两个流水线单元上，仅对 TCAM 与 HASH 两种资源的总量作了总量限制。折叠的两级 TCAM 总量不超过 1，HASH 总量不超过 3。则第 j 个基本块在折叠层中的资源限制条件表述为：

$$\sum_{i=0}^{n-1}C\mathrm{T}_i A_{ij}+\sum_{i=n}^{n-1}C\mathrm{T}_i A_{i,j+16}\le 1 \tag{10}$$

$$\sum_{i=0}^{n-1}CH_i A_{ij}+\sum_{i=0}^{n-1}CH_i A_{i,j+16}\le 3 \tag{11}$$

其中，由于折叠层仅发生在第 32 级之前，下标 j 满足：$0\le j\le 15$。

（3）TCAM 资源在流水线偶数级中限制约束：

流水线的偶数级：流水线从第 0 级开始排列,即流水线单元的序列是形如 $0,1,2,.....,n-1$ 的自然数列，TCAM 所在偶数级流水线单元指的是序列中偶数所代表的流水线单元。TCAM

资源在所有流水线中为偶数级资源总量约束为：

$$\sum_{i=0}^{n-1}\sum_{k=0}^{[n/2]} C\mathrm{T}_i A_{i,2k} \le 5 \tag{10}$$

（4）控制依赖关系约束：

基本块排布到流水线中需要满足基本块间的控制依赖关系，欲在 j 级流水线放第 i 个基本块。需要求解该基本块与剩下未排布的基本块的 $p, p \in \{0,1,2,\cdots i-1, i+1,\cdots,n-1\}$ 控制依赖关系。设基本块 p 放置于流水线上级数为 q ，则有:

$$A_{ij}(1-A_{ij}C_{pi}(j-q)) > 0 \tag{11}$$

式 11：$C_{pi}=0$ 表明 A_{ij} 只能取值 1，即基本块 i 可以排布在流水线 j 上使得剩余未排布的基本块 p 与基本块 i 不存在控制依赖关系。$C_{pi}=1$ 则表明 A_{ij} 可以取 1 也可以取 0。基本块 i 是否可以排布在流水线上需要进一步判断资源约束才可确定。

（5）数据依赖关系约束：

基本块排布到流水线中还需要满足基本块之间的数据依赖关系。欲在 j 级流水线放第 i 个基本块。需要求解该基本块与剩下未排布的基本块的 p 数据依赖关系。设基本块 p 放置于流水线上级数为 q ，则有:

$$A_{ij}(1-A_{ij}S_{pi}(j-q))(2-\mathrm{A}_{ij}S_{pi}(j-q)) > 0 \tag{12}$$

式 12 中：$S_{pi}=0$ 表明 A_{ij} 只能取值 1，即基本块 i 可以排布在流水线 j 上使得剩余未排布的基本块 p 与基本块 i 不存在数据依赖依赖关系。$S_{pi}=1$ 表明 A_{ij} 只能取 0，基本块 i 与基本块 p 之间存在读后写数据依赖。$S_{pi}=2$ 表明 A_{ij} 只能取 0，基本块 i 与基本块 p 之间存在写后操作数据依赖。

综合所有约束条件可得式 13 所示的约束集：

$$
\text{s.t.}\begin{cases}
\sum_{i=0}^{n-1} A_{ij} = 1, (j = 0,1,2,\cdots,n-1) \\
\sum_{i=0}^{n-1} CT_i a_{ij} \le 1 \\
\sum_{i=0}^{n-1} CH_i a_{ij} \le 2 \\
\sum_{i=0}^{n-1} CA_i a_{ij} \le 56 \\
\sum_{i=0}^{n-1} CQ_i a_{ij} \le 64 \\
\sum_{i=0}^{n-1} CT_i A_{ij} + \sum_{i=0}^{n-1} CT_i A_{i,j+16} \le 1, (0 \le j \le 15) \\
\sum_{i=0}^{n-1} CH_i A_{ij} + \sum_{i=0}^{n-1} CH_i A_{i,j+16} \le 3, (0 \le j \le 15) \\
\sum_{i=0}^{n-1} \sum_{k=0}^{[n/2]} CT_i A_{i,2k+1} \le 5 \\
A_{ij}(1 - A_{ij} C_{pi}(j-q)) > 0, (p \in \{0,1,2,\cdots i-1, i+1,\cdots,n-1\}, 0 \le j \le q) \\
A_{ij}(1 - A_{ij} S_{pi}(j-q))(2 - A_{ij} S_{pi}(j-q)) > 0, (p \in \{0,1,2,\cdots i-1, i+1,\cdots,n-1\}, 0 \le j < q)
\end{cases} \tag{13}
$$

6.1.2 PISA 芯片资源架构资源排布模型求解

求解最优化问题一般有梯度下降法、惩罚函数法、遗传算法、蚁群算法等[3]。在本文中由于资源排布时，需要动态考虑控制依赖关系、数据依赖关系、资源限制，上述算法实现难度较大。故考虑采用基于规则的启发式算法求解，启发式算法通常不能求得最优解，但求解大规模问题时，在运算时间上具有优势，且经过合理设计后可以得到满意解[4]。设计基于规则的启发式算法求解模型。给定数据的流图中，设流水线级数的不可达的下确界值为 m_1，即有 $L > m_1$。m_1 定义为仅考虑资源约束，不考虑基本块之间的控制、数据依赖约束时，能将所有基本块完全放置下的最小流水线级数。

（1） 启发式算法的求解步骤

Step1: 导入基本块间控制依赖关系矩阵 C 与基本块间数据依赖矩阵 S；

Step2: 随机将基本块分配到 n 条流水线中，由此初始化决策变量 A_{ij} 初始化 $K_{ij} = A_{ij}$ 并计算当前的 Z；

Step3: 依据基本块间控制依赖关系矩阵 C 与基本块间数据依赖矩阵 S 调整基本块分配方案，使分配方案满足 C 与 S 的约束。记录调整后的分配方案 K_{ij}，进行下一步；

Step4: 判断当前分配方案是否满足资源约束，若满足资源约束进行下一步，若不满足资源约束则转至 Step3；

Step5: 更新决策变量 A_{ij}，并计算目标函数 Z，计算流水线级数 L 并 $L > m_1$ 判断是否成立。

若 $L > m_1$ 成立则转至 Step3。若 $L > m_1$ 不成立则转至 Step6；

Step6: 输出满足资源约束、控制依赖约束、数据依赖约束的排布方案 A_{ij} 与流水线级数 L 与优化目标函数 Z 。

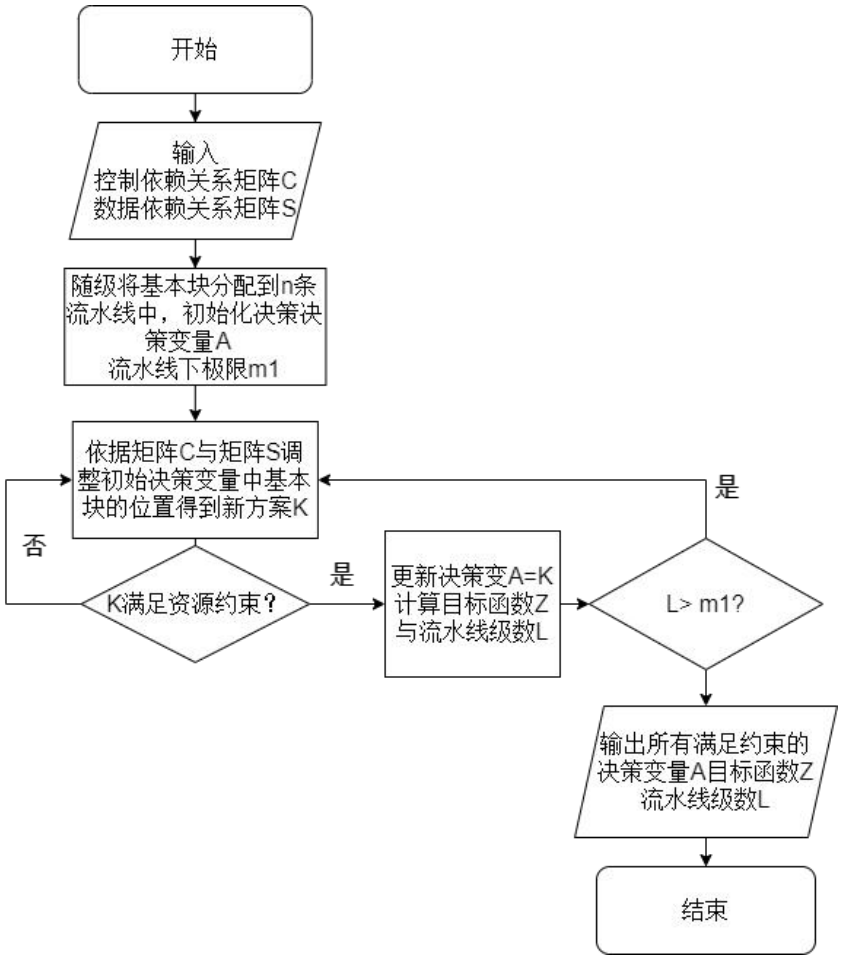

图 7 问题 1 的求解流程图

依据图 7 所示的求解流程图设计算法，在依据控制依赖矩阵和数据依赖矩阵调整基本块位置放置方案时。引入**近似单调队列概念，利用分治思想并使用归并排序算法排序，将排序后的结果进行反向扫描调整，**最终求解出具有数据依赖或控制依赖的基本块的**近似单调队列。**利用分治思想进行排序示例如图 8 所示。分析任意两个基本块 a 与 b 之间的控制依赖与数据依赖关系，有以下 5 种情形（为便于表述，以下使用 x_a, x_b 分别表时基本块 a 与基本块 b 所在的流水级）：

（1）基本块 a 与基本块 b 之间无任何依赖，则 x_a 与 x_b 无严格大小关系；

（2）基本块 a 与基本块 b 之间存在控制依赖但不存在数据依赖，则 $x_a \le x_b$ ；

（3）基本块 a 与基本块 b 之间不存在控制依赖且存在读后写数据依赖，则 $x_a \le x_b$ ；

（4）基本块 a 与基本块 b 之间存在控制依赖且存在读后写数据依赖，则 $x_a \le x_b$ ；

（5）基本块 a 与基本块 b 之间存在写后读数据依赖或写后写数据依赖，则 $x_a < x_b$；

综合上述情形可知，基本块 a 与 b 在流水线中除去情形(1)外，均有大小关系。因此将流图中的基本点分划分为两种：**第一种是该基本点与流图中剩余所有基本块均无控制依赖与数据依赖。第二种是该基本块与流图中剩余基本块至少一个基本块存在控制依赖与数据依赖中的至少一种。**针对第二种基本块构成的集合，利用分治思想，采用归并排序方式进行排序。

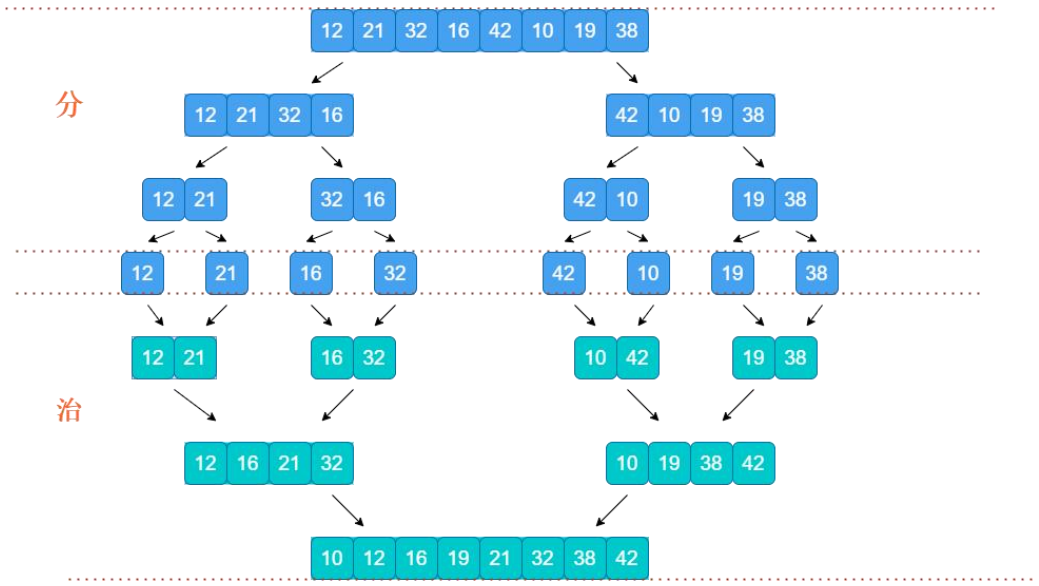

图 8 利用分治策略进行数据排序

反向扫描调整操作说明如下：

由归并排序算法求解得到结果可能会遗漏依赖约束，需要将排序后结果进行反向扫描调整后满足近似单调。举例说明如下：

由归并排序求解出的结果如图 9 所示。由图 9 中结果可知，当排序方向为从左至右时，由于由基本块构建的流图为有向图，由归并排序进行近似单调排序时，会遗漏图中所示的依赖约束条件 $D < C$。因此增加如图 10 所示的反向扫描操作，找到归并排序中遗漏的约束条件。结合归并排序的结果，加入遗漏依赖约束，调整排序结果。具体如图 11 所示，据遗漏的约束条件需将节点 C 与节点 D 位置对调，最终排序结果才是近似单调队列。

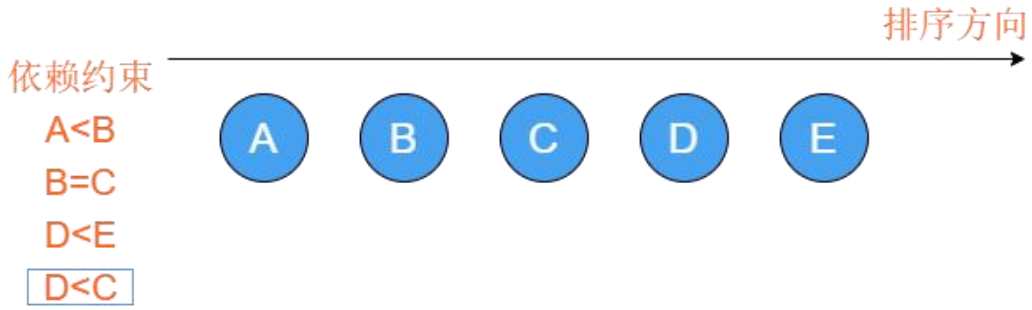

图 9 归并排序遗漏依赖约束示意图

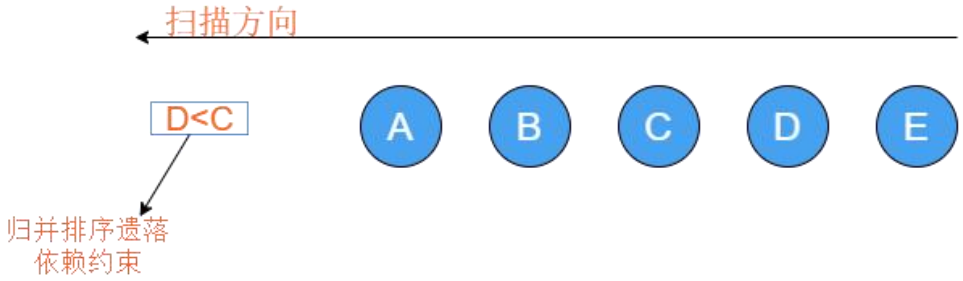

图 10 反向扫描查找遗漏约束条件

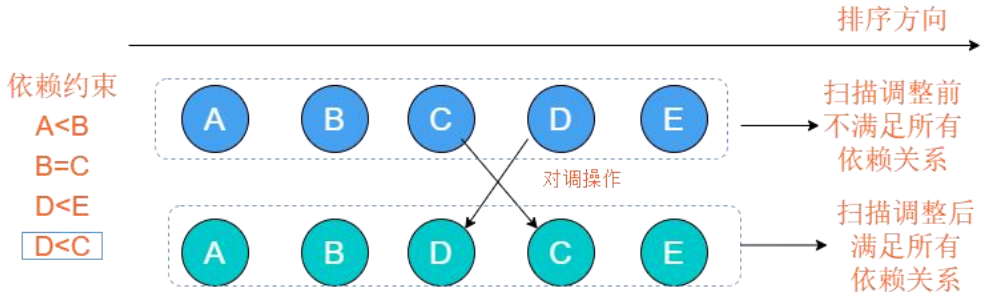

图 11 反向扫描查到的约束条件约束结果示意图

通过问题一模型建立与算法设计，求解得到**最小流水线级数为 48。**

6.2 问题二模型建立与求解

6.2.1 考虑执行流程的 PISA 芯片资源排布模型

问题二相对于问题一增加了“两个基本块位于同一执行流程”这一概念，并据此改变资源约束的约（2）、（3）、（5）。两个基本块位于同一执行流程的定义为“由其中一个基本块出发可到达另一个基本块，则这两个基本块位于同一执行流程。”依据定义可知，问题（2）的约束“流水线每级中同一条执行流程上的基本块的 HASH 资源之和最大为 2”表明流水线每级中可存在多条执行流程，每条执行流程上的基本块的 HASH 资源之和不超过 2。相比较问题一的约束“流水线每级的 HASH 资源最大为 2”而言，约束减弱。同理问题二的约束（3）、（5）对比问题的对应约束同为约束减弱后的结果。因此，问题二的约束空间包含问题一的约束空间。问题二流水线级数的最小值小于等于问题一种排布的流水线级数最小值。为此，考虑将问题的排布结果作为问题二的输入，构建有历史排布结果的单目标规划模型。模型的构建如下：

（一）决策变量

同问题一中决策变量一致，仍为 A_{ij}。A_{ij} 的初始值依据问题一的排布结果给出。

（二）目标函数

同问题一中的目标函数一致，仍为 Z。Z 的初始值依据问题一的排布结果给出。

（三）约束条件

位于同一流水线中的不同执行流程在满足资源约束情形下可共享资源，为求解出同一流水线中的所有不同执行流程，考虑深度优先搜索算法，求解流水线中的所有不同执行流程，算法伪代码如表 2 所示。

表 2 查找一级流水线中的所有执行流程

深度优先搜索法查找同级中所有位于同一执行流程基本块的伪代码

```
Input: 同一个流水线单元上的所有基本块，基本块之间的连通矩阵
Ouput:位于同一执行流程上的所有基本块组合方案
 定义全局变量列表     route_list, Qsum
利用同一流水线上所有基本块构造邻接链表 tree
Root 是当前数据节点
Dfs(tree,root,route_list,Qsum):
      For re in tree[root]:
          定义一个空列表 tmp_list
07.       If  节点 re 有子节点
08.                向 Qsum 中加入当前节点 re
09.                 递归调用 Dfs(tree,re,route_list,Qsum)
10.                 调用 Dfs 函数后从 Qsum 中取出节点 re
11.             else:
12.                往 tmp_list 中加入节点 re
13.       if   tmp_list 非空:
14.            结合 Qsum 与 tmp_list 给出当前路径
15.            在 route_list 中记录下当前路径
16.       return   记录有所有可能在同一流程上的基本块组合
```

设流水线 j 中的最大执行流程 HASH 资源之和表示为 $HASH_j$，最大执行流程的 ALU 资源之和表示为 ALU_j。则问题一的模型中约束（2）变为：

$$HASH_j \le 2 \tag{14}$$

问题一的模型中约束（3）变为：

$$ALU_j \le 56 \tag{15}$$

问题一的模型中折叠级的约束变为：

$$HASH_j + HASH_j \le 3 \tag{16}$$

$$\sum_{i=0}^{n-1} CT_i A_{ij} + \sum_{i=0}^{n-1} CT_i A_{i,j+16} \le 1 \tag{17}$$

综合所有约束条件得到式 18 所示的约束集：

$$
\text{s.t.}\begin{cases}
\sum_{i=0}^{n-1} A_{ij} = 1, (j = 0,1,2,\cdots,n-1) \\
\sum_{i=0}^{n-1} C\mathrm{T}_i a_{ij} \le 1 \\
HASH_j \le 2 \\
ALU_j \le 56 \\
\sum_{i=0}^{n-1} CQ_i a_{ij} \le 64 \\
\sum_{i=0}^{n-1} C\mathrm{T}_i A_{ij} + \sum_{i=0}^{n-1} C\mathrm{T}_i A_{i,j+16} \le 1, (0 \le \mathrm{j} \le 15) \\
HASH_j + HASH_{j+16} \le 3, (0 \le \mathrm{j} \le 15) \\
\sum_{i=0}^{n-1} \sum_{k=0}^{[n/2]} C\mathrm{T}_i A_{i,2k} \le 5 \\
A_{ij}(1 - A_{ij} C_{pi}(j-q)) > 0, (\mathrm{p} \in \{0,1,2,\cdots i-1, i+1,\cdots,\mathrm{n}-1\}, 0 \le \mathrm{j} \le q) \\
A_{ij}(1 - A_{ij} S_{pi}(j-q))(2 - \mathrm{A}_{ij}\, S_{pi}(j-q)) > 0, (\mathrm{p} \in \{0,1,2,\cdots i-1, i+1,\cdots,\mathrm{n}-1\}, 0 \le \mathrm{j} < q)
\end{cases} \tag{18}
$$

6.2.2 考虑执行流程的 PISA 芯片资源排布模型求解

给定数据的流图中，流水线级数的不可达的下确界值为m_2，即有$L \ge m_2$。m_2定义为仅考虑资源约束，不考虑基本块之间的控制、数据依赖约束时，能将所有基本块完全放置下的最小流水线级数。

（1）基于规则的启发式算法求解步骤

Step1: 导入基本块间控制依赖关系矩阵C、问题一的排布结果（决策变量）A_{ij}、基本块之间数据依赖矩阵S；

Step2: 初始化分配方案$K_{ij} = A_{ij}$，并由决策变量A_{ij}计算当前的流水线级数L与目标函数 Z；

Step3: 判断当前分配方案是否满足资源约束，若满足资源约束转至 Step5，若不满足资源约束则转至 Step4；

Step4: 依据基本块间控制依赖关系矩阵C与基本块间数据依赖矩阵S调整基本块分配方案使分配方案满足C与S的约束。记录调整后的分配方案K_{ij}，转至 Step3；

Step5: 更新决策变量A_{ij}，并计算目标函数 Z，计算流水线级数L并判断$L > m_2$是否成立。若$L > m_2$成立则转至 Step4。若$L > m_2$不成立则转至 Step6；

Step6: 输出满足资源约束、控制依赖约束、数据依赖约束的排布方案A_{ij}、流水线级数L

优化目标函数Z。

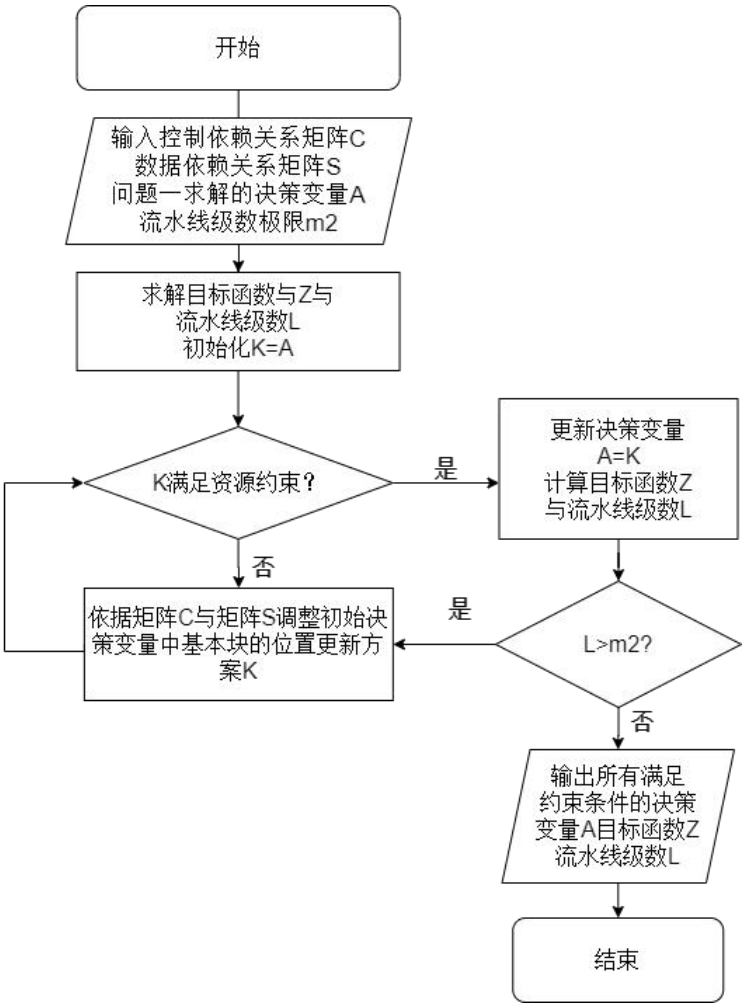

图 8 问题二的求解流程图

图 8 所示求解流程图中，依据控制依赖矩阵和数据依赖矩阵调整基本块位置放置方案时采用同问题一一致的求解算法。通过问题二模型建立与算法设计，求解得到**最小流水线级数为 32。**

七、模型的评价与推广

7.1 模型的优点

（1）本文针对 PISA 架构芯片资源排布问题进行研究，建立了单目标规划模型，并利用给定数据进行资源排布分配。模型求解结果表明，本文建立的模型能够有效的解决 PISA 架构芯片资源排布问题。

（2）建立的单目标优化模型解释性强，便于阅读理解。

（3）基于本文构建的模型，设计基于规则的启发式算法，有效降低模型求解复杂度，有效地增强了模型实用性。

7.2 模型的缺点

（1）建立的单目标规划模型对数据准确要求高，且求解该模型计算量大，高纬数据下难以进行快速地求解。
（2）设计的基于规则的启发式算法无法寻找到全局最优解，只能找到局部最优解。
（3）模型的约束进行了理论上的简化处理，将模型应用到实际情况中有所差别。

7.3 模型的推广

本文从 PISA 架构资源排布问题出发，综合考虑基本块资源排布时控制依赖、数据依赖、资源三个约束，使得模型具备一定的应用价值。其次，PISA 资源排布问题的建模对其他类似的有次序依赖的分配排布问题有一定的借鉴作用。

八、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] 余璟,岳振军,贾永兴.数据建模中的最优化方法探讨[J].科技世界，2021(30):66-67.DOI: 10.19694/j.cnki.issn2095-2457.2021.30.26.

[4] 华正明,杨丽静,刘振元. 使用启发式算法求解考虑资源可中途加入的资源投入问题[C]. 第 34 届中国控制与决策会议论文(7),2022:536-541.DOI:10.26914/c.cnkihy.2022.023676.

[5] 赖俊斌.基于共享内存的并行分组算法研究[D].华南理工大学,2015.

[6] 辛建芳.基于排队论的 D2D 异构蜂窝网络性能分析与优化[D].南京邮电大学,2020.DOI:10.27251/d.cnki.gnjdc.2020.001588.

九、附录

```
# 数据依赖预处理与有向无环图构建
import pandas as pd
import numpy as np
import math
import json
import graphviz as gz
def read_data(csv_file):
    largest_column_count =0
    with open(csv_file, 'r') as temp_f:
        lines = temp_f.readlines()
        for l in lines:
            column_count = len(l.split(','))
            #找到列数最多的行
            largest_column_count = column_count if largest_column_count < column_count
else largest_column_count
    temp_f.close()
    # colunm_names 为最大列数展开
    column_names = [str(i) for i in range(0, largest_column_count)]
    data = pd.read_csv(csv_file, header=None, delimiter=',', names=column_names)
    data.fillna(0)
    return data
# 读取数据
chipResourceData = pd.read_csv('./../../data/attachment1.csv',sep=',')
variableDependData = read_data('../../data/attachment2.csv')
blockRelationshipData = read_data('../../data/attachment3.csv')

def convert_int(data):
    data.fillna(-1, inplace=True)
    data[list(data.columns)] = data[list(data.columns)].astype(int)
    return data

blockRelationshipData = convert_int(blockRelationshipData)
variableDependData = variableDependData.fillna(-1)
blockRealtionshipMatrix = np.ndarray(shape=(607,607), dtype=int)
blockRelationshipData = blockRelationshipData.values.tolist()
edges = []
for i in blockRelationshipData:
    for j in range(1,12):
        if i[j] != -1:
            blockRealtionshipMatrix[i][j] = 1
            edges.append([str(i[0]), str(i[j])])
```

```
dot = gz.Digraph()
for index in range(607):
    dot.node(str(index), str(index))
dot.edges(edges)
dot.format = 'svg'
dot.render()

dataDependency = np.ndarray(shape=(607,607),dtype=int) # 无数据依赖 0， 读后写数据依赖 1，写后写数据依赖与写后读数据依赖均为 2
#查找 start 与 end 在有向图中是否有路径相连
def find(start,end,graph_dict):
    stack=[]
    stack.append(start)
    visited=[]
    visited.append(start)
    flag=False#false 表示 start 与 end 无路径连接
    while len(stack)>0:
        ver=stack.pop()
        if ver==end:
            flag=True
            break #start 与 end 之间存在路径,跳出 while 循环
        nodes=graph_dict[ver]
        for node in nodes:
            if node not in visited:
                stack.append(node)
                visited.append(node)
    return flag

def read_json(path):
    with open(path) as fp:
        data=json.load(fp)
    df={}
    for key,value in data.items():
        df[int(key)]=value
    return df

group_dict = read_json('./treeData.json')

basisExistence = np.ndarray(shape=(607,607),dtype=int) # 两个基本快之间的路径存在性
for i in range(len(basisExistence)):
    for j in range(len(basisExistence[i])):
        basisExistence[i][j] = int(find(i, j, group_dict))
np.save("../../data/基本块路径存在矩阵",basisExistence)
```

```
def get_block_variable_data(key):
    data =    variableDependData[variableDependData['0'].isin([key])]
    data.drop(['1'],axis=1,inplace=True)
    data =    data.values.tolist()
    for i in range(len(data)):
            while -1 in data[i]:
                data[i].remove(-1)
    return data
def get_dependency_data(rows, columns,order):
    res = 0 #数据依赖表示  0 为没有数据依赖  读后写依赖为 1  写后读依赖为 2
    befBlock = []
    aftBlock = []
    befBlock = get_block_variable_data(rows)
    aftBlock = get_block_variable_data(columns)
    if (set.intersection(set(befBlock[1]),set(aftBlock[0])) != set()): #  读后写
        res = 1
    if(set.intersection(set(befBlock[0]),set(aftBlock[1]))!=set())or
(set.intersection(set(befBlock[0]),set(aftBlock[1])) != set()): #  写后读或写后写
        res = 2
    return res
for rows in range(len(dataDependency)):
    for columns in range(len(dataDependency[rows])):
        if (rows == columns) or (basisExistence[rows][columns] == 0):
            dataDependency[rows][columns] = 0
        if basisExistence[rows][columns]:
            dataDependency[rows][columns]          =          get_dependency_data(rows,
columns,[rows,columns])
np.save("../../data/数据依赖矩阵", dataDependency)
```

问题一程序

```
#读取邻接表 attachment3.cav,创建链表,并存储在 json 文件中
import pandas as pd
import numpy as np
import csv
import json

#将读取的字符串数据转化为整型
def transform(datalab):
    data=[]
```

```
        for rep in datalab:
            tmp=[]
            for e in rep:
                tmp.append(int(e))
            data.append(tmp)
        return data

#读取数据
def read_data(path):
        datalab=0
        with open(path,'r') as fp:
            readline=csv.reader(fp)
            datalab=list(readline)
        data=transform(datalab)
        return data

#创建树结构
def creat_tree(data):
        tree={}
        for pop in data:
            key=pop[0]
            tmp=[]
            for i in range(1,len(pop)):
                tmp.append(pop[i])
            tree[key]=tmp
       return tree

#保存树结构
def write_data(path,name,data):
        with open(path+name,'w') as fp:
            json.dump(data,fp)
        return 0

#读取树结构
def read_json(path):
        with open(path) as fp:
            data=json.load(fp)
        df={}
        for key,value in data.items():
            df[int(key)]=value
        return df

# 数据集路径
path1=r'attachment3.csv'
```

```
data=read_data(path1)
#树链表存储路径
path2=r''
tree=creat_tree(data)
write_data(path2,'\树链表.json',tree)
print('树结构写入成功')

#回溯法处理控制依赖关系判断所有节点间的控制依赖关系并用矩阵存储
#0 表示两者不存在
#读取树结构
import numpy as np
import json

#查找 start  与 end  在有向图中是否有路径相连
def find(start,end,graph_dict):
    stack=[]
    stack.append(start)
    visited=[]
    visited.append(start)
    flag=False#false 表示 start 与 end 无路径连接
    while len(stack)>0:
        ver=stack.pop()
        if ver==end:
            flag=True
            break #start 与 end 之间存在路径,跳出 while 循环
        nodes=graph_dict[ver]
        for node in nodes:
            if node not in visited:
                stack.append(node)
                visited.append(node)
    return flag

def judge_control(start,end,graph_dict):
    if start==end:
        return 0
    Flag=False #标记多个孩子中每个都能到达目标点 end
    childM=graph_dict[start]#start 的孩子节点集合
    for pop in childM:#只有一个孩子节点的情况如何处理？
        stack=[]
        stack.append(pop)
        visited=[]
        visited.append(pop)
        flag=False#  标记单个孩子节点不存在到目标点的路径
        while len(stack)>0:
```

```
                ver=stack.pop()
                if ver==end:#当前节点存在路径到达目标点 end,退出该子树查找
                    flag=True
                    break
                nodes=graph_dict[ver]
                for node in nodes:
                    if node==end:
                        flag=True
                        break#找到目标点，跳出 for 循环
                    if node not in visited:
                        stack.append(node)
                        visited.append(node)
                if flag==True:#找到目标点跳出 while 循环
                    break
            #特殊情况，start 与 end 不相连，应判定为无依赖,但该情况包含了单个孩子节点无路径到达目标点的情况
            if flag==False: #有一个孩子节点无路径到达 end
                Flag=True
                break #跳出最外层 for 循环

#         if Flag==True:
#             return 1 #start 与 end 存在控制依赖关系

        if Flag==True:
            result=find(start,end,graph_dict)
            if result==False:
                return 0
            else:
                return 1 #start 与 end 存在控制依赖关系
    return 0

    #查找一个流水级中所有在同一执行流程
    class Find_process():
        def __init__(self,input1,data):
            self.input=input1#流水线上放置的基本块
            self.data=data#基本块之间的连接关系矩阵

        def transform(self,ver):
            tmp=[]
            for rep in self.input:
                if rep!=ver:
                    tmp.append(rep)
            return tmp
```

```
#打印出根节点到叶子节点的所有路径
def dfs(self,tree,root,route_list,Qsum):
    if len(tree[root])==0:
        return 0
    tmp_list=[]
    for i in tree[root]:#字典直接取值，取出的是键值，i 是 data 的键值
        if len(tree[i])>0:
            Qsum.append(i)
            dfs(tree,i,route_list,Qsum)
            Qsum.pop()
        else:
            tmp_list.append(i)
    if tmp_list:#该句执行，说明 tmp_list 已装入 data 的所有元素
            route_list.append(Qsum+tmp_list)
    return route_list

def find_max_route(self):#data 是判断是否连通的
    test_tree={}
    for pop in self.input:
        childs=self.transform(pop)#以 pop 为根节点，其余节点为子节点的树
        test_tree[pop]=childs

    for i in test_tree:
        nodes=test_tree[i]
        tmp=[]
        for node in nodes:
            if self.data[i][node]==1:
                tmp.append(node)
        test_tree[i]=tmp

    route_dict={}#存储输入的每个节点的所有路径
    for rep in self.input:
        Qsum=[rep]#先存入根节点
        route_list=[]
        self.dfs(test_tree,rep,route_list,Qsum)
        route_dict[rep]=route_list
    return route_dict
```

#创建一个函数 judge,输入 A，B 两基本块，在函数内部使用控制依赖与数据依赖对 A，B 的控制、数据依赖关系进行判断，输出三种数值 0，1，2

```
    #0 表示 A，B 既无控制依赖也无数据依赖关系
    #judge(A,B)==1,则 A<=B
    #judge(A,B)==2,则有 A<B
    import numpy as np

    class relate_judge():
        def __init__(self,path1,path2):
            self.data=np.load(path1)#控制关系
            self.developdata=np.load(path2)#数据依赖关系

        def judge(self):
            row,col=self.data.shape
            tmpdata=np.zeros((row,col))
            for i in range(row):
                for j in range(row):
                    if i!=j and self.developdata[i][j]==0 and self.data[i][j]==0:
                        tmpdata[i][j]=0
                        continue
                    if i!=j and self.developdata[i][j]==2:
                        tmpdata[i][j]=2
                        continue
                    tmpdata[i][j]=1
            for i in range(row):
                for j in range(row):
                    if i==j:
                        tmpdata[i][j]=3
            return tmpdata

import numpy as np
#筛选出既无控制也无数据依赖关系的基本块
def select_block(data):
    block0=[]#不需要进行放置顺序判断的基本块
    block1=[]#需要进行放置顺序判断的基本块
    row,col=data.shape
    for i in range(row):
        tmp0=0
        for j in range(col):
            if i!=j and data[i][j]==0:
                tmp0=tmp0+1
        if tmp0==row-1:
            block0.append(i)
        else:
            block1.append(i)
    return block0,block1
```

```
block0,block1=select_block(relation_data)
print('无控制、无数据依赖关系的基本块个数为{}'.format(len(block0)))
print('存在控制、数据依赖关系至少一个的基本块个数为{}'.format(len(block1)))

求解近似单调队列
#将有序存在的基本块序列，转化为一个近似单调增的序列
#后续改进为输入基本块序列，处理后返回两个结果，返回一个近似单调增序列，和一个无序要求的序列
import numpy as np
class single_transform():
    def __init__(self,block,relation_data):
        self.relation=relation_data#任意两者之间的先后放置顺序判断矩阵
        self.data=block#待排列的序列

    def quicksort(self,left,right):
        if left>right:
            return 0
#           tmp=[]
#           for re in range(left,right+1):
#               tmp.append(self.data[re])
#           print('原数列为{}'.format(self.data))
#           print('当前处理列为{}'.format(tmp))
        Lpointer=left
        Rpointer=right
        rep=self.data[left]
        while Lpointer!=Rpointer:
            while self.relation[self.data[Rpointer]][rep]==0 and Lpointer<Rpointer:
#self.relation[rep][self.data[Rpointer]]==0
#                   print('L={},R={}'.format(Lpointer,Rpointer))
                Rpointer=Rpointer-1

            while self.relation[self.data[Lpointer]][rep]>0 and Lpointer<Rpointer:
#                   self.relation[rep][self.data[Lpointer]]==0
#                   print('L={},R={}'.format(Lpointer,Rpointer))
                Lpointer=Lpointer+1

            if Lpointer<Rpointer:
#                   print('原序列为{}'.format(tmp))
#                   print('L!=R 时发生的交换')
                a=self.data[Lpointer]
                self.data[Lpointer]=self.data[Rpointer]
                self.data[Rpointer]=a
#                   print('交换后的序列为{}\n'.format(self.data))
```

```
            #Lpointer 与 Rpointer 相遇的时候进行下列操作
#               print('原序列为{}'.format(tmp))
#               print('L={},R={}'.format(Lpointer,Rpointer))
#               print('L==R 发生的交换')
            self.data[left]=self.data[Rpointer]
            self.data[Rpointer]=rep


            self.quicksort(Rpointer+1,right)
            self.quicksort(left,Lpointer-1)
            return 0

        #检验基本块最终的近似单调序列是否符合要求
        def find(self,now_index):#relation_data,查找 now 元素是否符合条件，
            sequence=self.data
            tmp=[]
            flag=False#假设 now 元素与在最后序列中与其它所有元素不冲突
            #前向比较
            for i in range(0,now_index):
                if self.relation[sequence[now_index]][sequence[i]]:
                    flag=True
                    tmp.append(sequence[i])
            #后向比较
            for i in range(now_index+1,len(sequence)):
                if self.relation[sequence[i]][sequence[now_index]]:
                    flag=True
                    tmp.append(sequence[i])
            return flag,tmp

        #扫描结果，并对调存在逆序的位置
        def scan_result(self,input1):
#               data=input1[::-1]
            data=input1
            for i in range(len(data)):
                for j in range(i+1,len(data)):

                    if self.relation[data[i]][data[j]]>0:
                        a=data[i]
                        data[i]=data[j]
                        data[j]=a
            tmp=data[::-1]
            self.data=tmp
            return tmp
```

```
    def seq_find(self):
        sequence=self.data
        false_set={}#存放发生冲突位置上的元素
        for rep in range(len(sequence)):
            tmp,tmpset=self.find(rep)
            if tmp:
                false_set[sequence[rep]]=tmpset
        if len(false_set)==0:
            print('输入序列所有元素不存在先后顺序错乱')
        else:
            print('输入序列存在元素出现先后顺序错乱')
        return false_set

问题一分发基本块的程序
#按近似单调的序列 block1,逐个向流水线上放置基本块
#资源限制函数：

#创建流水线资源空间
def create_source(num):
    Line={}
    for i in range(num):
        tmp={'T':[],'H':[],'A':[],'Q':[]}
        Line[i]=tmp
    return Line

#创建流水线的基本块存储空间
def create_space(num):
    tree={}
    for i in range(num):
        tree[i]=[]
    return tree

#折叠层资源计算,line 之前的所有折叠层的资源总量
def sum_one_source(storage_source,line):
    line_source=[]
    line_tree=storage_source[line]
    for rep in line_tree.keys():
        line_source.append(sum(line_tree[rep]))
    return line_source

#计算两条流水线的对应资源总和
def sum_two_line(storage_source,line1,line2):
    tmp1=sum_one_source(storage_source,line1)
    tmp2=sum_one_source(storage_source,line2)
```

```
    tmp=[]
    for i in range(len(tmp1)):
        tmp.append(tmp1[i]+tmp2[i])
    return tmp

#计算一个块加入流水线 line 后所得资源总和，输入块的资源列表
def line_block(storage_source,line,block_list):
    tmp1=sum_one_source(storage_source,line)
    tmp=[]
    for i in range(len(tmp1)):
        tmp.append(tmp1[i]+block_list[i])
    return tmp

#统计加入块在 line 层后，line 及 line 之前有 TCAM 的偶数层总数
def sum_double(storage_source,storage_space,line,block_list):
    #找当前 storage_space 中最大非空流水线单元
    tmp=[]
    for key,value in storage_space.items():
        if len(value)>0:
            tmp.append(key)
    SUM=max(tmp)
    double=[]#当前最大非空流水线单元之前的的偶数层
    for i in range(SUM+1):
        if i%2==0:
            double.append(i)
    Qsum=0#统计有 TCAM 的流水线单元级数
    for k in double:
        tmp=sum_one_source(storage_source,k)
        if tmp[0]==1:
            Qsum=Qsum+1
    return Qsum

#输入流水线级数 line，基本块 block，判断加入后是否超过资源限制
def judge_source(storage_source,storage_space,line,block):#block 是一个列表存放块所需资源

    #1.先判断四种资源是否分别满足
   #tmp=[T,H,A,Q],依次存放
    Flag=True

    #计算新加块与 line 流水线的资源总和
    btmp=line_block(storage_source,line,block)
    #加入块后的四种资源约束条件判断
    if btmp[0]>1 or btmp[1]>2 or btmp[2]>56 or btmp[3]>64:
        Flag=False
```

```
    #2.折叠层 TCAM 与 HASH 资源限制,32 层以下受到折叠级数的影响
    if line<32 and line>=16:
        rep=sum_two_line(storage_source,line,line-16)
        tmp=[]
        for op in range(len(rep)):
            tmp.append(rep[op]+block[op])
        if tmp[0]>1 or tmp[1]>3:
            Flag=False

    #3.有 TCAM 的偶数层不超过 5
    #统计 line 及 line 之前，有 TCAM 的偶数层级数
    if line%2==0 and btmp[0]==1:
        Qsum=sum_double(storage_source,storage_space,line,block)
        if Qsum>5:
            Flag=False
    return Flag

#将 re 基本块放入第 line 级流水线中
def storage(storage_source,storage_space,re,line,block_tree):
    #放入资源
    source=storage_source[line]#目标流水单元
    data=block_tree[re]#re 模块的资源
    key=list(source.keys())
    for i in range(4):
        storage_source[line][key[i]].append(data[i])
    #放入基本块
    storage_space[line].append(re)
    return 0

#放置基本块的函数,放入块，返回块在流水线中的布置链表
#输入基本块列表，基本块的资源占用列表
#无序放置从 0 层开始查找，有序放置从当前层开始往后放置，
def share_block(storage_source,storage_space,block,block_tree,flag):#flag=True,放置的是有序
块
    route={}
    K=1
    if flag:
        Qline=0#line 标记放置一个块后的，最大的流水线单元级数
        #实现块放置流水线单元自动跳转
        for re in block:
            while True:
                pop=judge_source(storage_source,storage_space,Qline,block_tree[re])#判断
re 能否放入 Qline 层
```

```
                    if pop:
                        storage(storage_source,storage_space,re,Qline,block_tree)
                        K=K+1
                        route[K]=[re,Qline]
                        break
                    else:
                        Qline=Qline+1
        else:
            for re in block:
                Qline=0
                while True:
                    pop=judge_source(storage_source,storage_space,Qline,block_tree[re])#判断
re 能否放入 line 层
                    if pop:
                        storage(storage_source,storage_space,re,Qline,block_tree)
                        K=K+1
                        route[K]=[re,Qline]
                        break
                    else:
                        Qline=Qline+1

        print('资源表插入记录{}'.format(storage_source))
        print('记录表插入记录{}'.format(storage_space))
        return route

#测试分发案例
import numpy as np
import random
#读取近似单调基本序列：
sequence=np.load(r'近似单调基本块序列.npy')
block0=np.load(r'无序基本块序列.npy')
#读取基本块的四种资源约束数据
source=read_json(r'基本块的四种资源约束数据.json')
#创建两张邻接标，一张记录流水线的资源状况，一张是流水线的基本块放置记录
n=607#测试中，暂时先以一个序列长度赋值给 n
storage_source=create_source(n)
storage_space=create_space(n)
#打乱无序基本块
random.shuffle(block0)
route1=share_block(storage_source,storage_space,sequence,source,True)#先分发有序基本块到
流水线上
route2=share_block(storage_source,storage_space,block0,source,False)#再分发无序基本块到
流水线上
print('各流水线单元的资源占用链表：{}\n'.format(storage_source))
```

print('各流水线单元的基本块放置情况链表：{}'.format(storage_space))

问题 2 求解程序

```
#读取邻接表 attachment3.cav,创建链表,并存储在 json 文件中
import pandas as pd
import numpy as np
import csv
import json

#将读取的字符串数据转化为整型
def transform(datalab):
    data=[]
    for rep in datalab:
        tmp=[]
        for e in rep:
            tmp.append(int(e))
        data.append(tmp)
    return data

#读取数据
def read_data(path):
    datalab=0
    with open(path,'r') as fp:
        readline=csv.reader(fp)
        datalab=list(readline)
    data=transform(datalab)
    return data

#创建树结构
def creat_tree(data):
    tree={}
    for pop in data:
        key=pop[0]
        tmp=[]
        for i in range(1,len(pop)):
            tmp.append(pop[i])
        tree[key]=tmp
    return tree

#保存树结构
def write_data(path,name,data):
    with open(path+name,'w') as fp:
        json.dump(data,fp)
    return 0
```

```
#读取树结构
def read_json(path):
    with open(path) as fp:
        data=json.load(fp)
    df={}
    for key,value in data.items():
        df[int(key)]=value
    return df

# 数据集路径
path1=r'attachment3.csv'
data=read_data(path1)
#树链表存储路径
path2=r'D 题处理结果数据集'
tree=creat_tree(data)
write_data(path2,'\树链表.json',tree)
print('树结构写入成功')

#回溯法处理控制依赖关系判断所有节点间的控制依赖关系并用矩阵存储
#0 表示两者不存在
#读取树结构
import numpy as np
import json

#查找 start 与 end 在有向图中是否有路径相连
def find(start,end,graph_dict):
    stack=[]
    stack.append(start)
    visited=[]
    visited.append(start)
    flag=False#false 表示 start 与 end 无路径连接
    while len(stack)>0:
        ver=stack.pop()
        if ver==end:
            flag=True
            break #start 与 end 之间存在路径,跳出 while 循环
        nodes=graph_dict[ver]
        for node in nodes:
            if node not in visited:
                stack.append(node)
                visited.append(node)
    return flag
```

```
def judge_control(start,end,graph_dict):
    if start==end:
        return 0
    Flag=False #标记多个孩子中每个都能到达目标点 end
    childM=graph_dict[start]#start 的孩子节点集合
    for pop in childM:#只有一个孩子节点的情况如何处理？
        stack=[]
        stack.append(pop)
        visited=[]
        visited.append(pop)
        flag=False#  标记单个孩子节点不存在到目标点的路径
        while len(stack)>0:
            ver=stack.pop()
            if ver==end:#当前节点存在路径到达目标点 end,退出该子树查找
                flag=True
                break
            nodes=graph_dict[ver]
            for node in nodes:
                if node==end:
                    flag=True
                    break#找到目标点，跳出 for 循环
                if node not in visited:
                    stack.append(node)
                    visited.append(node)
            if flag==True:#找到目标点跳出 while 循环
                break
        #特殊情况，start  与 end 不相连，应判定为无依赖,但该情况包含了单个孩子节点无路径到达目标点的情况
        if flag==False: #有一个孩子节点无路径到达 end
            Flag=True
            break #跳出最外层 for 循环

#       if Flag==True:
#           return 1 #start  与 end 存在控制依赖关系

    if Flag==True:
        result=find(start,end,graph_dict)
        if result==False:
            return 0
        else:
            return 1 #start  与 end 存在控制依赖关系
    return 0

    #查找一个流水级中所有在同一执行流程
```

```
class Find_process():
    def __init__(self,input1,data):
        self.input=input1#流水线上放置的基本块
        self.data=data#基本块之间的连接关系矩阵

    def transform(self,ver):
        tmp=[]
        for rep in self.input:
            if rep!=ver:
                tmp.append(rep)
        return tmp

    #打印出根节点到叶子节点的所有路径
    def dfs(self,tree,root,route_list,Qsum):
        if len(tree[root])==0:
            return 0
        tmp_list=[]
        for i in tree[root]:#字典直接取值，取出的是键值，i 是 data 的键值
            if len(tree[i])>0:
                Qsum.append(i)
                dfs(tree,i,route_list,Qsum)
                Qsum.pop()
            else:
                tmp_list.append(i)
        if tmp_list:#该句执行，说明 tmp_list 已装入 data 的所有元素
                route_list.append(Qsum+tmp_list)
        return route_list

    def find_max_route(self):#data 是判断是否连通的
        test_tree={}
        for pop in self.input:
            childs=self.transform(pop)#以 pop 为根节点，其余节点为子节点的树
            test_tree[pop]=childs

        for i in test_tree:
            nodes=test_tree[i]
            tmp=[]
            for node in nodes:
                if self.data[i][node]==1:
                    tmp.append(node)
            test_tree[i]=tmp

        route_dict={}#存储输入的每个节点的所有路径
        for rep in self.input:
```

```
                Qsum=[rep]#先存入根节点
                route_list=[]
                self.dfs(test_tree,rep,route_list,Qsum)
                route_dict[rep]=route_list
            return route_dict

    #judge(A,B)==1,则 A<=B
    #judge(A,B)==2,则有 A<B
    import numpy as np

    class relate_judge():
        def __init__(self,path1,path2):
            self.data=np.load(path1)#控制关系
            self.developdata=np.load(path2)#数据依赖关系

        def judge(self):
            row,col=self.data.shape
            tmpdata=np.zeros((row,col))
            for i in range(row):
                for j in range(row):
                    if i!=j and self.developdata[i][j]==0 and self.data[i][j]==0:
                        tmpdata[i][j]=0
                        continue
                    if i!=j and self.developdata[i][j]==2:
                        tmpdata[i][j]=2
                        continue
                    tmpdata[i][j]=1
            for i in range(row):
                for j in range(row):
                    if i==j:
                        tmpdata[i][j]=3
            return tmpdata

import numpy as np
#筛选出既无控制也无数据依赖关系的基本块
def select_block(data):
    block0=[]#不需要进行放置顺序判断的基本块
    block1=[]#需要进行放置顺序判断的基本块
    row,col=data.shape
    for i in range(row):
        tmp0=0
        for j in range(col):
            if i!=j and data[i][j]==0:
                tmp0=tmp0+1
```

```
            if tmp0==row-1:
                block0.append(i)
            else:
                block1.append(i)
        return block0,block1
block0,block1=select_block(relation_data)
print('无控制、无数据依赖关系的基本块个数为{}'.format(len(block0)))
print('存在控制、数据依赖关系至少一个的基本块个数为{}'.format(len(block1)))

#将有序存在的基本块序列，转化为一个近似单调增的序列
#后续改进为输入基本块序列，处理后返回两个结果，返回一个近似单调增序列，和一个无序要求的序列
import numpy as np
class single_transform():
    def __init__(self,block,relation_data):
        self.relation=relation_data#任意两者之间的先后放置顺序判断矩阵
        self.data=block#待排列的序列

    def quicksort(self,left,right):
        if left>right:
            return 0
#           tmp=[]
#           for re in range(left,right+1):
#               tmp.append(self.data[re])
#           print('原数列为{}'.format(self.data))
#           print('当前处理列为{}'.format(tmp))
        Lpointer=left
        Rpointer=right
        rep=self.data[left]
        while Lpointer!=Rpointer:
            while self.relation[self.data[Rpointer]][rep]==0 and Lpointer<Rpointer:
#self.relation[rep][self.data[Rpointer]]==0
#                   print('L={},R={}'.format(Lpointer,Rpointer))
                Rpointer=Rpointer-1

            while self.relation[self.data[Lpointer]][rep]>0 and Lpointer<Rpointer:
#                   self.relation[rep][self.data[Lpointer]]==0
#                   print('L={},R={}'.format(Lpointer,Rpointer))
                Lpointer=Lpointer+1

            if Lpointer<Rpointer:
#                   print('原序列为{}'.format(tmp))
#                   print('L!=R 时发生的交换')
                a=self.data[Lpointer]
```

```
                    self.data[Lpointer]=self.data[Rpointer]
                    self.data[Rpointer]=a
#                       print('交换后的序列为{}\n'.format(self.data))

        #Lpointer 与 Rpointer 相遇的时候进行下列操作
#           print('原序列为{}'.format(tmp))
#           print('L={},R={}'.format(Lpointer,Rpointer))
#           print('L==R 发生的交换')
        self.data[left]=self.data[Rpointer]
        self.data[Rpointer]=rep



        self.quicksort(Rpointer+1,right)
        self.quicksort(left,Lpointer-1)
        return 0

    #检验基本块最终的近似单调序列是否符合要求
    def find(self,now_index):#relation_data,查找 now 元素是否符合条件，
        sequence=self.data
        tmp=[]
        flag=False#假设 now 元素与在最后序列中与其它所有元素不冲突
        #前向比较
        for i in range(0,now_index):
            if self.relation[sequence[now_index]][sequence[i]]:
                flag=True
                tmp.append(sequence[i])
        #后向比较
        for i in range(now_index+1,len(sequence)):
            if self.relation[sequence[i]][sequence[now_index]]:
                flag=True
                tmp.append(sequence[i])
        return flag,tmp

    #扫描结果，并对调存在逆序的位置
    def scan_result(self,input1):
#           data=input1[::-1]
        data=input1
        for i in range(len(data)):
            for j in range(i+1,len(data)):

                if self.relation[data[i]][data[j]]>0:
                    a=data[i]
                    data[i]=data[j]
                    data[j]=a
```

```
            tmp=data[::-1]
            self.data=tmp
            return tmp

        def seq_find(self):
            sequence=self.data
            false_set={}#存放发生冲突位置上的元素
            for rep in range(len(sequence)):
                tmp,tmpset=self.find(rep)
                if tmp:
                    false_set[sequence[rep]]=tmpset
            if len(false_set)==0:
                print('输入序列所有元素不存在先后顺序错乱')
            else:
                print('输入序列存在元素出现先后顺序错乱')
            return false_set

分发基本块的程序
#按近似单调的序列 block1,逐个向流水线上放置基本块
#资源限制函数：

#创建流水线资源空间
def create_source(num):
    Line={}
    for i in range(num):
        tmp={'T':[],'H':[],'A':[],'Q':[]}
        Line[i]=tmp
    return Line

#创建流水线的基本块存储空间
def create_space(num):
    tree={}
    for i in range(num):
        tree[i]=[]
    return tree

#折叠层资源计算,line 之前的所有折叠层的资源总量
def sum_one_source(storage_source,line):
    line_source=[]
    line_tree=storage_source[line]
    for rep in line_tree.keys():
        line_source.append(sum(line_tree[rep]))
    return line_source
```

```
#计算两条流水线的对应资源总和
def sum_two_line(storage_source,line1,line2):
    tmp1=sum_one_source(storage_source,line1)
    tmp2=sum_one_source(storage_source,line2)
    tmp=[]
    for i in range(len(tmp1)):
        tmp.append(tmp1[i]+tmp2[i])
    return tmp

#计算一个块加入流水线 line 后所得资源总和，输入块的资源列表
def line_block(storage_source,line,block_list):
    tmp1=sum_one_source(storage_source,line)
    tmp=[]
    for i in range(len(tmp1)):
        tmp.append(tmp1[i]+block_list[i])
    return tmp

#统计加入块在 line 层后，line 及 line 之前有 TCAM 的偶数层总数
def sum_double(storage_source,storage_space,line,block_list):
    #找当前 storage_space 中最大非空流水线单元
    tmp=[]
    for key,value in storage_space.items():
        if len(value)>0:
            tmp.append(key)
    SUM=max(tmp)
    double=[]#当前最大非空流水线单元之前的的偶数层
    for i in range(SUM+1):
        if i%2==0:
            double.append(i)
    Qsum=0#统计有 TCAM 的流水线单元级数
    for k in double:
        tmp=sum_one_source(storage_source,k)
        if tmp[0]==1:
            Qsum=Qsum+1
    return Qsum

#输入流水线级数 line，基本块 block，判断加入后是否超过资源限制
def judge_source(storage_source,storage_space,line,block):#block 是一个列表存放块所需资源
    #1.先判断四种资源是否分别满足
   #tmp=[T,H,A,Q],依次存放
    Flag=True

    #计算新加块与 line 流水线的资源总和
    btmp=line_block(storage_source,line,block)
```

```
        #加入块后的四种资源约束条件判断
        if btmp[0]>1 or btmp[1]>2 or btmp[2]>56 or btmp[3]>64:
            Flag=False

        #2.折叠层 TCAM 与 HASH 资源限制,32 层以下受到折叠级数的影响
        if line<32 and line>=16:
            rep=sum_two_line(storage_source,line,line-16)
            tmp=[]
            for op in range(len(rep)):
                tmp.append(rep[op]+block[op])
            if tmp[0]>1 or tmp[1]>3:
                Flag=False

        #3.有 TCAM 的偶数层不超过 5
        #统计 line 及 line 之前，有 TCAM 的偶数层级数
        if line%2==0 and btmp[0]==1:
            Qsum=sum_double(storage_source,storage_space,line,block)
            if Qsum>5:
                Flag=False
        return Flag

#将 re 基本块放入第 line 级流水线中
def storage(storage_source,storage_space,re,line,block_tree):
        #放入资源
        source=storage_source[line]#目标流水单元
        data=block_tree[re]#re 模块的资源
        key=list(source.keys())
        for i in range(4):
            storage_source[line][key[i]].append(data[i])
        #放入基本块
        storage_space[line].append(re)
        return 0

#放置基本块的函数,放入块，返回块在流水线中的布置链表
#输入基本块列表，基本块的资源占用列表
#无序放置从 0 层开始查找，有序放置从当前层开始往后放置，
def share_block(storage_source,storage_space,block,block_tree,flag):#flag=True,放置的是有序
块
        route={}
        K=1
        if flag:
            Qline=0#line 标记放置一个块后的，最大的流水线单元级数
            #实现块放置流水线单元自动跳转
            for re in block:
```

```
                while True:
                    pop=judge_source(storage_source,storage_space,Qline,block_tree[re])#判断
re 能否放入 Qline 层
                    if pop:
                        storage(storage_source,storage_space,re,Qline,block_tree)
                        K=K+1
                        route[K]=[re,Qline]
                        break
                    else:
                        Qline=Qline+1
        else:
            for re in block:
                Qline=0
                while True:
                    pop=judge_source(storage_source,storage_space,Qline,block_tree[re])#判断
re 能否放入 line 层
                    if pop:
                        storage(storage_source,storage_space,re,Qline,block_tree)
                        K=K+1
                        route[K]=[re,Qline]
                        break
                    else:
                        Qline=Qline+1

        print('资源表插入记录{}'.format(storage_source))
        print('记录表插入记录{}'.format(storage_space))
        return route

#测试分发案例
import numpy as np
import random
#读取近似单调基本序列：
sequence=np.load(r'C:\Users\张松鸿\Desktop\华为杯\D 题\D 题处理结果数据集\近似单调基
本块序列.npy')
block0=np.load(r'C:\Users\张松鸿\Desktop\华为杯\D 题\D 题处理结果数据集\无序基本块序
列.npy')
#读取基本块的四种资源约束数据
source=read_json(r'C:\Users\张松鸿\Desktop\华为杯\D 题\D 题处理结果数据集\基本块的四
种资源约束数据.json')

#创建两张邻接标，一张记录流水线的资源状况，一张是流水线的基本块放置记录
n=607#测试中，暂时先以一个序列长度赋值给 n
storage_source=create_source(n)
storage_space=create_space(n)
```

```
#打乱无序基本块
random.shuffle(block0)
route1=share_block(storage_source,storage_space,sequence,source,True)#先分发有序基本块到流水线上
route2=share_block(storage_source,storage_space,block0,source,False)#再分发无序基本块到流水线上
print('各流水线单元的资源占用链表：{}\n'.format(storage_source))
print('各流水线单元的基本块放置情况链表：{}'.format(storage_space))

问题二分发基本块的文件
import pandas as pd
import numpy as np
import json
import csv
f=open(r'attachment1.csv')
chipResourceData = pd.read_csv(f,sep=',') #读取资源消耗数据
blockRelationMartix = np.load(r'基本块路径存在矩阵.npy')

#读取树结构
def read_json(path):
    with open(path) as fp:
        data=json.load(fp)
    df={}
    for key,value in data.items():
        df[int(key)]=value
    return df

#查找一个流水级中所有在同一执行流程
class Find_process():
    def __init__(self,input1,data):
        self.input=input1#流水线上放置的基本块
        self.data=data#基本块之间的连接关系矩阵

    def transform(self,ver):
        tmp=[]
        for rep in self.input:
            if rep!=ver:
                tmp.append(rep)
        return tmp

    #打印出根节点到叶子节点的所有路径
    def dfs(self,tree,root,route_list,Qsum):
        if len(tree[root])==0:
            return 0
```

```
            tmp_list=[]
            for i in tree[root]:#字典直接取值，取出的是键值，i 是 data 的键值
                if len(tree[i])>0:
                    Qsum.append(i)
                    self.dfs(tree,i,route_list,Qsum)
                    Qsum.pop()
                else:
                    tmp_list.append(i)
            if tmp_list:#该句执行，说明 tmp_list 已装入 data 的所有元素
                    route_list.append(Qsum+tmp_list)
            return route_list

        def find_max_route(self):#data 是判断是否连通的
            test_tree={}
            for pop in self.input:
                childs=self.transform(pop)#以 pop 为根节点，其余节点为子节点的树
                test_tree[pop]=childs

            for i in test_tree:
                nodes=test_tree[i]
                tmp=[]
                for node in nodes:
                    if self.data[i][node]==1:
                        tmp.append(node)
                test_tree[i]=tmp

            route_dict={}#存储输入的每个节点的所有路径
            for rep in self.input:
                Qsum=[rep]#先存入根节点
                route_list=[]
                self.dfs(test_tree,rep,route_list,Qsum)
                route_dict[rep]=route_list
            return route_dict

    def judge_two_new(storage_space) :
        judgeList = []
        flowchart = []#  所有层的最大执行流程 HASH 资源与 ALU 资源集合列表
    [[HASH1,ALU1],[HASH2,ALU2]]
        for _,value in storage_space.items():
            if value == []:
                flowchart.append([0,0])
            else:
                flowchart.append(hierarchyBlock_flowchart(value))
```

```
    # HASH 资源约束判断 ALU 资源约束判断
    for j in range(len(flowchart)):
        judgeList.append(flowchart[j][1] <= 56)
        if (j >= 0) & (j <= 15):
            judgeList.append((flowchart[j][0]+flowchart[j+16][0]) <= 3)
        else:
            judgeList.append(flowchart[j][0] <= 2)
    return not(False in judgeList)


def hierarchyBlock_flowchart(hierarcy): # 传入级的所有块，返回该层的所有流程与所有流程中的 HASH、ALU 资源数目之和与最大流程资源和的 maxHASH 与 maxALU
    allProcess = Find_process(hierarcy, blockRelationMartix)
    res = allProcess.find_max_route()
    flowchatList = [] # 同一流水线中的所有执行流程 [[a,b],[c,d]]形式
    for key, value in res.items():
        if value != []:
            for flow in value:
                flowchatList.append(flow)
    HASH = []
    ALU = []
    for flow in flowchatList:
        temp = [0,0]
        for block in flowchatList:
            temp[0] += chipResourceData["HASH"][block]
            temp[1] += chipResourceData["ALU"][block]
        HASH.append(temp[0])
        ALU.append(temp[1])
    return [max(HASH),max(ALU)]


#按近似单调的序列 block1,逐个向流水线上放置基本块
#资源限制函数：
#创建流水线资源空间
def create_source(num):
    Line={}
    for i in range(num):
        tmp={'T':[],'H':[],'A':[],'Q':[]}
        Line[i]=tmp
    return Line

#创建流水线的基本块存储空间
def create_space(num):
    tree={}
```

```
    for i in range(num):
        tree[i]=[]
    return tree

#折叠层资源计算,line 之前的所有折叠层的资源总量
def sum_one_source(storage_source,line):
    line_source=[]
    line_tree=storage_source[line]
    for rep in line_tree.keys():
        line_source.append(sum(line_tree[rep]))
    return line_source

#计算两条流水线的对应资源总和
def sum_two_line(storage_source,line1,line2):
    tmp1=sum_one_source(storage_source,line1)
    tmp2=sum_one_source(storage_source,line2)
    tmp=[]
    for i in range(len(tmp1)):
        tmp.append(tmp1[i]+tmp2[i])
    return tmp

#计算一个块加入流水线 line 后所得资源总和，输入块的资源列表
def line_block(storage_source,line,block_list):
    tmp1=sum_one_source(storage_source,line)
    tmp=[]
    for i in range(len(tmp1)):
        tmp.append(tmp1[i]+block_list[i])
    return tmp

#统计加入块在 line 层后，line 及 line 之前有 TCAM 的偶数层总数
def sum_double(storage_source,storage_space,line,block_list):
    #找当前 storage_space 中最大非空流水线单元
    tmp=[]
    for key,value in storage_space.items():
        if len(value)>0:
            tmp.append(key)
    SUM=max(tmp)
    double=[]#当前最大非空流水线单元之前的的偶数层
    for i in range(SUM+1):
        if i%2==0:
            double.append(i)
    Qsum=0#统计有 TCAM 的流水线单元级数
    for k in double:
        tmp=sum_one_source(storage_source,k)
```

```
            if tmp[0]==1:
                Qsum=Qsum+1
        return Qsum

#将 re 基本块放入第 line 级流水线中
def storage(storage_source,storage_space,re,line,block_tree):
        #放入资源
        source=storage_source[line]#目标流水单元
        data=block_tree[re]#re 模块的资源
        key=list(source.keys())
        for i in range(4):
            storage_source[line][key[i]].append(data[i])
        #放入基本块
        storage_space[line].append(re)
        return 0

#弹出不满足条件的基本块
def re_storage(storage_source,storage_space,line):
        tmp=storage_source[line]
        for key in tmp.keys():
            storage_source[line][key].pop()
        storage_space[line].pop()
        return 0

#输入流水线级数 line，基本块 block，判断加入后是否超过资源限制
def judge_source(storage_source,storage_space,re,line,block_tree):#block 是一个列表存放块所需资源

        #1.先判断四种资源是否分别满足
       #tmp=[T,H,A,Q],依次存放
        Flag=True

        #计算新加块与 line 流水线的资源总和
        tmp=line_block(storage_source,line,block_tree[re])

        #加入块后的四种资源约束条件判断
        if tmp[0]>1 or tmp[3]>64:
            Flag=False

        #3.有 TCAM 的偶数层不超过 5
        #统计当前最大流水线单元之前，拥有 TCAM 的且层级为偶数的流水线单元
        if line%2==0 and tmp[0]==1:
            Qsum=sum_double(storage_source,storage_space,line,block_tree[re])
            if Qsum>5:
```

```
                Flag=False
    return Flag

    #问题二新增条件：
    storage(storage_source,storage_space,re,Qline,block_tree)
    pop=judge_two_new(storage_space)
    if not pop:
        Flag=False
        re_storage(storage_source,storage_space,line)

    return Flag

#放置基本块的函数,放入块，返回块在流水线中的布置链表
#输入基本块列表，基本块的资源占用列表

#无序放置从 0 层开始查找，有序放置从当前层开始往后放置，
def share_block(storage_source,storage_space,block,block_tree,flag):
    #flag=True,放置的是有序块
    if flag:
        Qline=0#line 标记放置一个块后的，最大的流水线单元级数
        #实现块放置流水线单元自动跳转
        for re in block:
            while True:
                pop=judge_source(storage_source,storage_space,re,Qline,block_tree)# 判 断
re 能否放入 Qline 层
                if pop:
                    storage(storage_source,storage_space,re,Qline,block_tree)
                    K=K+1
                    route[K]=[re,Qline]
                    break
                else:
                    Qline=Qline+1
    else:
        for re in block:
            Qline=0
            while True:
                pop=judge_source(storage_source,storage_space,re,Qline,block_tree)# 判 断
re 能否放入 line 层
                if pop:
                    storage(storage_source,storage_space,re,Qline,block_tree)
                    break
                else:
                    Qline=Qline+1
```

```
    print('资源表插入记录{}'.format(storage_source))
    print('记录表插入记录{}'.format(storage_space))
    return 0


#以下为程序调用部分
#读取近似单调基本序列：
sequence=np.load(r'近似单调基本块序列.nblock0=np.load(r'无序基本块序列.npy')
#读取基本块的四种资源约束数据
source=read_json(r'基本块的四种资源约束数据.json')

#创建两张邻接标，一张记录流水线的资源状况，一张是流水线的基本块放置记录
n=607#测试中，暂时先以一个序列长度赋值给 n
storage_source=create_source(n)
storage_space=create_space(n)
route1=share_block(storage_source,storage_space,sequence,source,True)#先分发有序基本块到流水线上
route2=share_block(storage_source,storage_space,block0,source,False)#再分发无序基本块到流水线上
print('各流水线单元的资源占用链表：{}\n'.format(storage_source))
print('各流水线单元的基本块放置情况链表：{}'.format(storage_space))
```