


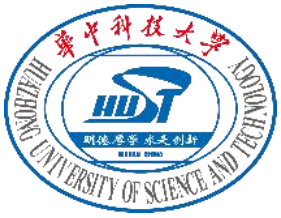

中国研究生创新实践系列大赛

中国光谷·“华为杯”第十九届中国研究生数学建模竞赛

学　　校	西北工业大学
参赛队号	22106990096
队员姓名	1.高亦菲 2.李一玮 3.曲雪

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　　D　　**PISA 架构芯片资源排布研究**

摘　　要：

随着全球“芯”缺浪潮的持续发酵，作为“工业粮食”的芯片技术成为我国亟待突围的产业之一。PISA 作为兼具良好处理速度与可编程性的交换芯片架构，有效缓解了传统固定功能的交换芯片研发效率低下的问题。为充分发挥芯片能力，高资源利用率的芯片资源排布算法的选择显得尤其重要。鉴于此，本文将复杂的基本块资源排布问题拆解为多个子问题，并结合各类约束条件，通过建立**目标规划模型**与**基本块排布模型**逐步求解，然后设计**启发式规则**对求解结果进行优化，最终得到的尽可能短的流水线级数作为问题的解。

针对问题一，本文将其划分为**依赖关系分析**和**基本块资源排布**两个子问题。**针对子问题一**，分析处理附件 attachment3.csv 数据得到**控制流图 CFG** 与控制流图的**前向支配树 FDT**，CFG 和 FDT **差集**即为基本块的控制依赖关系。其次，读取附件 attachment2.csv 中基本块的读写关系，利用**广度优先搜索算法 BFS** 判断基本块间的连通性来确定数据依赖关系。将控制依赖与数据依赖关系取并集得到完整的依赖关系。**针对子问题二**，将完成的依赖关系抽象为**带权有向无环图**，初步确定基本块在流水线中排布的层级关系。然后，以占用尽量短的流水线级数为主要目标，结合 attachment1.csv 中的基本块资源需求和给定的资源约束条件建立**目标规划模型**。模型的求解设计了**基本块排布模型**，依据串行排布的原则，从无前序依赖关系的基本块中依据**随机规则**选取基本块放入最早可放入的流水线级，直到所有基本块完成排布。

问题一得出流水线占用级数为 **40 级**。

针对问题二，由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，我们在问题一的基础上引入不在一条执行流程上的基本块的概念，缓解了问题一中对 HASH 和 ALU 的占用问题。本文将问题二拆解为**判断两个基本块是否在同一条执行流程**、**基本块资源排布**两个子问题。**针对子问题一**，根据程序流程图，利用**广度优先搜索算法**判断两个基本块是否在一条执行流程上，即从一个基本块出发是否可到达另一个基本块，获得**基本块执行流程关系对称矩阵**。**针对问题二**，同样以占用尽量短的流水线级数为主要目标，结合不在同一执行流程上的基本块可共享 HASH 和 ALU 资源这一条件与更改后的三个约束，建立与问题一类似的**目标规划模型**。求解时对于有新的基本块排布的流水线，**合并**该级流水线中与其处于不同控制流的基本块对 HASH 和 ALU 的**资源需求**。

问题二得出流水线占用的级数为 **34 级**。

求解方案优化考虑到随机规则选取的不稳定性，对**基本块选取规则**进行优化，本文构建了最早开始时间 EST、最多紧后资源块 MST、最早开始时间且最多紧后资源块 EST_MST 等 **6 种启发式规则**，依据启发式规则选取要加入流水线的基本块。**本文综合了 10 次随机规则和 6 种启发式规则求解的最优结果作为最终的流水线排布方案。**优化后，问题一得出流水线占用级数为 **40 级**。问题二得出流水线占用的级数为 **34 级**。

最后，本文发现了问题一的**性能瓶颈**在于 HASH 资源，问题二的**性能瓶颈**在于 TCAM 资源。通过综合考虑控制依赖、数据依赖和各级流水线的资源约束条件，证明了所提方案的有效性与普适性。总结了所提算法与模型的优缺点，并对未来研究工作进行了展望。

关键词：PISA；资源利用率；基本块资源排布问题；目标规划模型；随机算法；启发式算法

目录

1 问题重述

1.1 问题背景

在全球“芯”缺浪潮持续发酵的形势下，作为“工业粮食”、安全根本的芯片是各个大国必争的高科技技术，是我国亟待突围的产业之一。传统固定功能的交换芯片，随网络协议的更新而需重新设计，这大大降低了研发效率，为此，诞生了可编程的交换芯片。兼具良好处理速度与可编程性的交换芯片架构 PISA（Protocol Independent Switch Architecture），是当前主流之一。PISA 架构包括报文解析、多级的报文处理流水线与报文重组三个部分。本课题在只关注多级报文处理流水线部分的前提下，解决 PISA 架构芯片资源排布问题，即由 P4 语言得到的 P4 程序在经编译器编译时，首先会被划分为系列基本块，随后将各基本块排布到流水线各级当中。为充分发挥芯片能力，需考虑多种约条件束，找到高资源利用率的资源排布算法。

1.2 问题提出

题目给出 attachment1.csv、attachment2.csv、attachment3.csv 三个附件，分别表示了各基本块使用的资源信息、各基本块读写的变量信息、各基本块在流程图种的邻接基本块信息。充分挖掘各基本块间的数据依赖、控制依赖关系，建立相应的资源排布模型及算法，进一步解决以下问题：

问题一：结合给定的资源约束条件，实现资源排布并优化，使占用的流水线级数尽量短，以求最大化芯片资源利用率，要求输出基本块排布结果。

问题二：数据流图中，不在同一条执行流程上的基本块（从一个基本块出发不可达另一个基本块），可以共享 HASH 资源和 ALU 资源，即不在同一条执行流程上的基本块，任一个的 HASH 资源与 ALU 资源均不超过每级资源限制，就可排布到同一级。结合题目更改后的资源约束条件，实现资源排布并优化，使占用的流水线级数尽量短，并输出基本块排布结果。

2 模型假设

[1] 假设实验场景均位于理想状态，不受温度和电磁等因素影响。
[2] 假设实验过程中不存在由芯片故障导致的误差。
[3] 假设只关注 PISA 架构芯片多级的报文处理流水线部分。
[4] 假设每级流水线除题目所给的资源约束外，其他条件均相同。
[5] 假设可排布的流水线级数没有上限。
[6] 假设随机模型种取随机数的过程是完全随机的。

3 符号说明

本文中所用符号的说明如下表所示。

符号	说明	备注
GW	带权有向无环图	
$weight(u,v)$	表示(u,v)偏序关系	0:小于等于，1:严格小于
B	基本块集合	
d_{uk}	基本块 u 对资源 k 的需求	
P	基本块分布矩阵	N×N 方阵
p_{ui}	基本块 u 放在第 i 级流水线	
R	流水线资源使用矩阵	4×N
r_{kj}	第 j 级流水线对资源 k 的使用	
M	流水线级数序号集合	
L_i	基本块 i 可放入的最早流水线级数	i=0...N
DIN_i	基本块 i 的入度	i=0...N
C	矩阵$(1,0,1,0,...)^T$	N×1
q_{uv}	有向无环图中有向边(u,v)的权值	
E	流程约束矩阵	N×N 方阵
G	基本块邻接关系图	N×N 方阵
GF	基本块邻接关系逆序图	N×N 方阵
$idomGF$	基本块的前向支配图	N×N 方阵

4 问题一的模型建立与求解

4.1 问题分析

问题一要求考虑各基本块间的数据依赖、控制依赖关系情况下，结合给定的资源约束条件，实现基本块资源排布并优化，使占用的流水线级数尽量短，以求最大化芯片资源利用率。题目给出 attachment1.csv、attachment2.csv、attachment3.csv 三个附件分别为各基本块使用的资源信息、各基本块读写的变量信息以及各基本块在流程图种的邻接基本块信息。

通过对题目分析，我们得出本题可拆解为以下**三个子问题**：

子问题一：通过设计算法与模型，对附件二、附件三中的数据进行处理分析，确定各基本块间的控制依赖关系与数据依赖关系，从而初步确定基本块在流水线中排布的层级关系。

子问题二：结合题目所给资源约束条件与附件一的资源使用数据，建立模型，确定基本块在流水线中的级数排布情况。

子问题三：对基本块排布进行优化，使得基本块占用的流水线级数尽可能短。

4.2 模型建立与求解

4.2.1 依赖分析模型

为初步确定基本块在流水线中排布的层级关系，我们首先对附件二 attachment2.csv 与附件三 attachment3.csv 中的数据进行了预处理，分别得到具有数据依赖的基本块对集合与有控制依赖关系的基本块对集合。在此基础上，我们对两个集合进行合并，得到基本块偏序关系集合，即流水线级数关系分别为严格小于和小于等于的两个列表。模型建立流程如图 4-1 所示。

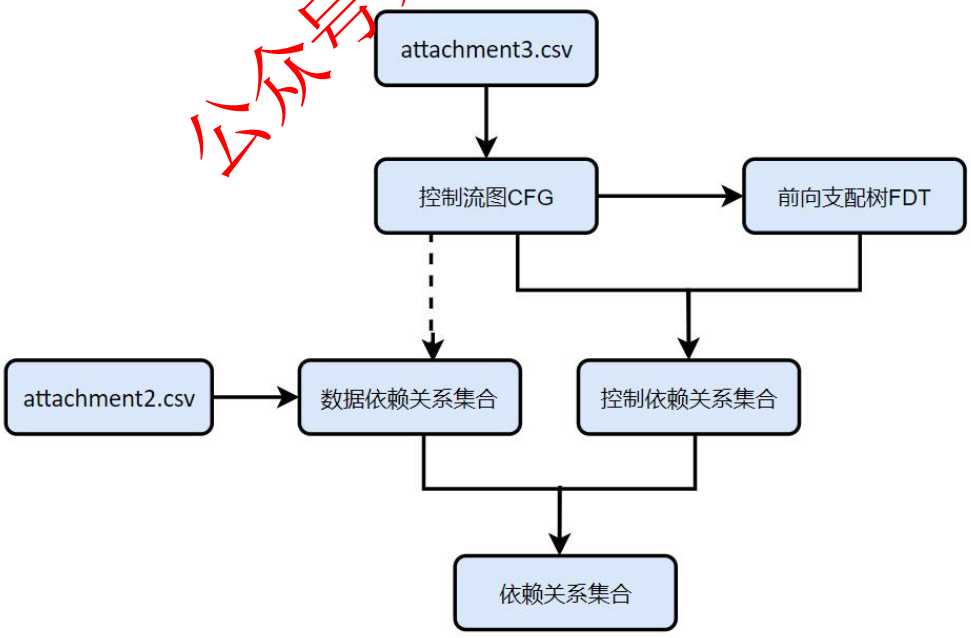

图 4-1 依赖分析模型流程图

模型实现伪代码如表 4-1 所示。

表 4-1

确定依赖关系伪代码

输入： attachment1.csv，attachment2.csv，attachment3.csv

输出： dependence_L，dependence_LE

```
attachment3.csv→linkList #存取各基本块在程序流程图中的邻接关系
linkList→G，GF #将邻接关系及其逆序转化为图
GF+根节点→idomGF #获取邻接关系的前项支配图
for G 中的每条边(i, j)：
    if (i, j) ∉idomGF：
        (i, j)→control_dependance_less_equal #存入小于等于的控制依赖列表
    end
attachment2.csv→wr_dict #提取基本块与相关变量的读写关系
wr_dict→ddg_dict #将以基本块为键值的字典转化为以变量为键值的字典
for ddg_dict 中的每个变量：
    if 该变量对应的基本块对(i, j)之间存在通路：
    if (i, j)存在写后写，写后读关系:
        (i, j)→data_dependance_less #存入严格小于的数据依赖列表
    end
    if (i, j)存在读后写关系:
        (i, j)→data_dependance_less_equal #存入小于等于的数据依赖列表
    end
  end
all_less_equal = control_dependance_less_equal ∪ data_dependance_less_equal
all_less_equal→dependence_LE
data_dependance_less→dependence_L
```

（1）控制依赖分析

控制依赖是程序控制流导致的一种约束。设 A 和 B 为某程序中的两个节点，若 B 能否执行取决于 A 的执行结果，则称 B 控制依赖于 A。通俗地讲，若节点 B 控制依赖于节点 A，则要求 A 与 B 之间可达，A 至少有两个直接后继节点。因此，若节点 B 是节点 A 的唯一直接后继，则 B 并不控制依赖于 A。PISA 架构中，若基本块 A 与基本块 B 存在控制依赖，则 A 排布的流水线级数需要小于或等于 B 排布的流水线级数。

方法思想： 附件三给出各基本块在流程图中的邻接基本块信息，我们需建立数学模型以分析各基本块间的控制依赖关系，从而得到它们在流水线中排布的级数关系。首先，

我们将附件三数据信息转换为控制流图 CFG(Control Flow Graph)。其次，反转控制流图 CFG，并添加根结点，进一步求得反向控制流图的支配树，即为 CFG 的前向支配树 FDT(Forward dominance Tree)。最后，我们查找在 CFG 中，但不在 FDT 中的边，得到存在控制依赖关系的基本块集合 S。

1) 控制流图 CFG

控制流图（程序流程图）也即一个有向图，当程序被划分为基本块后，若将基本块视为一个基本单元节点，在程序执行流程上互为前驱和后继关系的两个基本块之间可视作存在一条有向边，整个程序就能转换为一个有向图 G。

以表 4-2 数据集为例，我们可将其转化为如图 4- 2 所示控制流图。

表 4-2

基本块序号	邻接基本块序号
0	1
1	2，3
2	4，5
3	5，7
4	6
5	6
6	7
7	8
8	

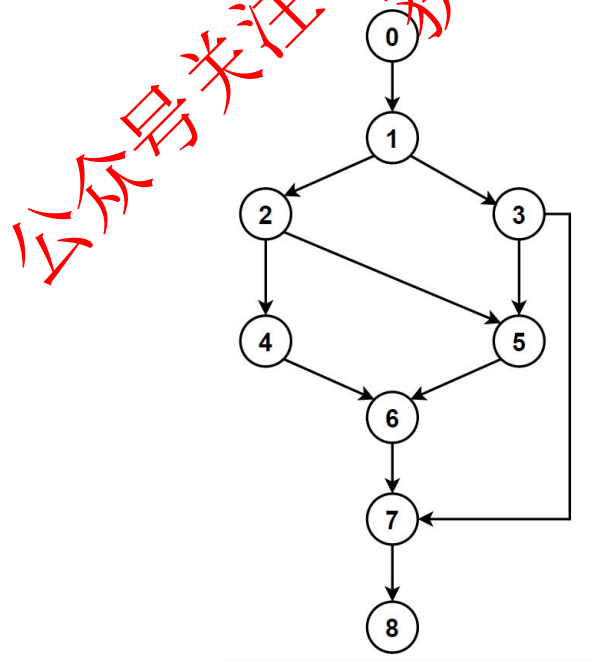

图 4-2 控制流图 CFG

2) 前向支配树 FDT

对于有向图 G, 设起点为 s，终点为 t，s -> t 之间可能存在多条路径，所有从 s 到 t 的路径中都必经的点，我们称之为支配点。删除该点，将不再存在由 s 可达 t 的路径，因此称起点为 s 时，该点支配了 t。s -> t 中可能有许多支配点，由这些支配点构成的树，称为

支配树。反向拓扑的支配树称为前向支配树（FDT），即以根节点为函数出口的支配树。将图 4-2 控制流图可转换为如图 4-3 所示前向支配书 FDT。

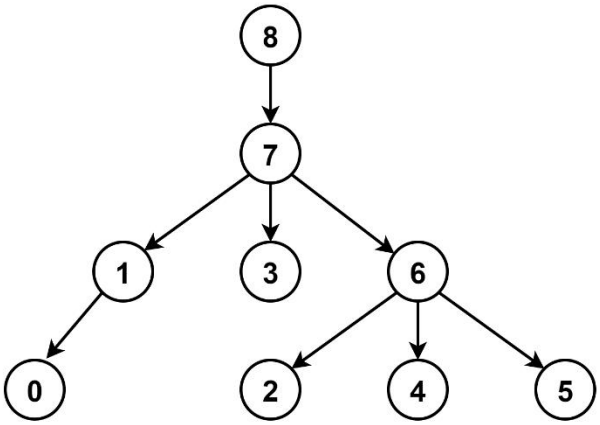

图 4-3 前向支配树 FDT

3) 控制依赖数据集合 S

将控制流图 CFG 与前向支配树 FDT 结合，找到在 FDT 中而不在 CFG 中的边所对应的节点对，得到控制依赖图 CDG，如图 4-4 所示。构成的集合即为具有控制依赖关系的基本块集合 control_dependance_less_equal={(1,2),(1,3),(2,4),(2,5),(3,5)}。

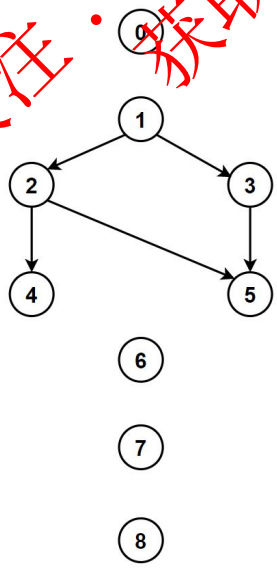

图 4-4 控制依赖图 CFG

（2）数据依赖分析模型

数据依赖是数据流之间的一种约束，具体包含写后读、读后写及写后写三种依赖方式。在 PISA 架构中，若基本块 A 和 B 存在写后读或写后写数据依赖，基本块 A 排布的流水线级数需小于基本块 B 排布的级数，若基本块 A 和 B 存在读后写数据依赖，基本块 A 排布的流水线级数需小于或等于基本块 B 排布的级数。

方法思想：附件二给出各基本块读写的变量信息，同样，我们需建立数学模型分析各基本块间的数据依赖关系，进而得到它们排布的流水线级数关系。首先，我们读

取附件二中基本块的读写关系，并将基本块读取状态按变量分组。其次，分别对每个变量中的基本块两两判断是否连通，若连通，进一步分析数据依赖关系，即判断是写后读、读后写还是写后写。

1) 可变容器存储模型

我们使用了一个可变容器模型字典 ddg_dict 来存储 attachment2.csv 中获取的存储关系。字典的键为所有的变量，值为每个变量对应的基本块及其对该变量的读写情况，例如：{'X1': [('0', 'R'), ('1', 'W'), ('3', 'R'), ('2', 'W')], 'X2': [('0', 'W')]}表示基本块 0，1，3，2 分别对 X1 变量有读，写，读，写的操作，基本块 0 对 X2 有读操作。通过字典，我们可以将 607 个基本块的庞大数据集，根据访问的变量划分为 824 个小数据集，检查数据依赖时，只需检查对应小数据集内各基本块对之间的读写状态即可。

2) 广度优先搜索算法

由于两个基本块 u，v 产生数据依赖的前提是：在程序流程图 G 中 u，v 之间需存在通路，所以在检查读写状态之前，需要检查顶点对(u,v)在有向无环图 G 中的可达性，我们在此使用的是广度优先搜索算法。广度优先搜索 BFS(Breadth-First Search)是连通图的一种遍历算法，可以系统地展开并检查图中的所有节点，直到找到结果或所有节点均被访问，具体伪代码如表 4-3 所示。

表 4-3

广度优先搜索算法伪代码

```
输入：G，u，v
输出：True or False
准备一个空闲的队列 q
将 u 放入 q 中
标记 u 为已访问
while (q 不为空)：
    访问 q 中队首元素 x
    删除 x
    for ( x 所有后继节点)：
      if (该点未被访问过且合法)：
      将该点加入队列末尾
      if (该节点是 v)：
        return True      # 找到目标节点
          break;
        end
       end
return False   # 队列为空，搜索结束
```

若在可达性检查中得知基本块 u 到基本块 v 是可达的，则需要读取字典 ddg_dict 中 u 和 v 对变量的读写情况，若 u 为读取，v 为写入，则根据数据依赖规则，(u,v)应加入小于

等于的数据依赖关系列表 data_dependance_less_equal 中，若 u 为写入，v 为读取，或 u 为写入，v 为写入，(u,v)应加入严格小于的数据依赖关系列表 data_dependance_less 中。

（3）依赖关系图模型

图论是研究点、线间关系的一门学科，属于应用数学的一部分。现实生活中，凡是涉及到事物间的关系，都可以抽象为图论模型。据于此，我们可以将上述求得的控制依赖关系与数据依赖关系映射到一个带权有向无环图中。

由于 4.2.1 得到的控制依赖全部都是小于等于的偏序关系，可以将 data_dependance_less_equal 与 control_dependance_less_equal 取并集去除重复项。至此，我们可以获取到由所有控制依赖和数据依赖条件提取出的两组流水线级数偏序关系列表，偏序关系为小于等于的基本块对列表 dependence_LE 和偏序关系为小于的基本块对列表 dependence_L。

我们用点表示基本块，连线表示基本块之间的读写关系，如果连线存在，表示两端的基本块对相同的变量有读写关系。为进一步区分读写关系的“读后写”“写后读”“写后写”，我们构造了带权的有向图，刻画事物间的联系（比如城市间的距离），就给连线赋一个权值，从而抽象成赋值图。

4.2.2 基本块排布模型

（1）模型建立

对流水线基本块分配问题进行需求分析，抽象问题并建立数学模型。建模过程需要考虑资源约束条件，优化目标等属性及基本块相互间的依赖关系，为更好的通过数学语言描述问题，我们建立如下**变量**：

$R = \{k | k = 0,1,2,3\}$:资源集合，其中 *0*:TCAM,1:HASH,3:ALU,*4*:QUALIFY

$B = \{u | u = 0,\ldots,N\}$, N=606 : 基本块集合

$M = \{i | i = 0,\ldots,N\}$: 流水线级数序号集合

L_i：基本块 i 所在流水线级数列表

d_{uk}：基本块 u 对资源 k 的需求

p_{ui}：基本块 u 放在第 i 级流水线

r_{ki}：第 i 级流水线对资源 k 的使用

$$m_i = \begin{cases} 1, & \text{该级有基本块} \\ 0, & \text{该级无基本块} \end{cases}$$

$q_{u,v}$：有向无环图中有向边(u,v)的权值

$$q_{u,v} = \begin{cases} 1, & \text{基本块 u 流水线级数小于基本块 v} \\ 0, & \text{基本块 u 流水线级数小于等于基本块 v} \end{cases}$$

结合题目描述，**目标函数**可表述为在满足基本块依赖关系与流水线各级资源限制的情况下，尽量减少流水线的级数，可以抽象为：

$$min\{\sum_{i=0}^{N} m_i\} \qquad (1)$$

根据题目要求，还需以下**约束条件**：

1) 依赖关系约束

基本块排布时需考虑它们之间的控制依赖与数据依赖关系，从而确定它们所在的流水线层级关系。若有向无环图 GW 中有向边权值为 1，则两个基本块在流水线中排布的级数为严格小于关系，若有向边权值为 0，则两个基本块在流水线中排布的级数为小于等于关系，由此可以抽象为：

$$L_u < L_v,\ q_{u,v} = 1 \quad (2)$$

$$L_u \le L_v,\ q_{u,v} = 0 \quad (3)$$

2) 资源约束

设 i 级流水线包含的基本块集为 M(i)，则第 i 级流水线对第 k 种资源的需求量为：

$$r_{ki} = \sum d_{ik},\ \forall i \in M(i)$$

由题目给出条件建立约束：

$$r_{0,i} \le 1 \quad (4)$$

$$r_{1,i} \le 2 \quad (5)$$

$$r_{2,i} \le 5 \quad (6)$$

$$r_{3,i} \le 64 \quad (7)$$

3) 折叠约束

题目规定流水线第 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。若所需的流水线级数超过 32 级，则从第 32 开始的级数不考虑折叠资源限制。据此，可抽象为：

$$r_{0,i} + r_{0,i+16} \le 1,\ i=0...15 \quad (8)$$

$$r_{1,i} + r_{1,i+16} \le 3,\ i=0...15 \quad (9)$$

4) TCAM 偶数级约束

设矩阵 C=(1,0,1,0...)^T，则约束“有 TCAM 资源的偶数级数量不超过 5”可以抽象为：

$$\sum_{i=0}^{N} (r_{0,i} * C) \le 5 \quad (10)$$

5) 基本块只能排到一级流水线约束

题目要求每个基本块只能排到一级流水线约束，可以抽象为：

$$\forall i,\ \sum_{i=0}^{N} p_{ui} = 1 \quad (11)$$

综上所述，我们可以得到问题一的目标函数模型，如下所示：

$$min\{\sum_{i=0}^{N} m_i\} \quad (1)$$

$$\text{s.t.} \begin{cases} L_u < L_v,\ q_{u,v} = 1 \\ L_u \le L_v,\ q_{u,v} = 0 \\ r_{0,i} \le 1 \\ r_{1,i} \le 2 \\ r_{2,i} \le 5 \\ r_{3,i} \le 64 \\ r_{0,i} + r_{0,i+16} \le 1,\ i = 0...15 \\ r_{1,i} + r_{1,i+16} \le 3,\ i = 0...15 \\ \sum_{i=0}^{N} (r_{0,i} * C) \le 5 \\ \forall i,\ \sum_{i=0}^{N} p_{ui} = 1 \end{cases} \quad (12)$$

（2）模型求解与优化

通过建立依赖关系模型，对附件 attachment2.csv 与 attachment3.csv 中数据预处理后，得到基本块在流水线中排布的初步依赖关系，流水线排布过程中对所有基本块进行如下规定：

- 前序基本块：两个基本块的依赖关系中需要优先排布的块；
- 后序基本块：两个基本块的依赖关系中需要靠后排布的块；
- FB：完成流水线排布的基本块集合；
- CB：所有前序基本块都处于 FB 的基本块集合；
- NB：未完成流水线排布的基本块集合。

依据串行进度生成机制，优先排布前序基本块全部放入流水线的基本块，即集合 CB 中的基本块。CB 是一系列无前序依赖关系的基本块集合，不断从中**随机**选取一个基本块，计算其最早可排入的流水线级数，并将其排布该级流水线，并更新依赖关系 CB 与基本块集合 CB 和 NB。NB 为空说明所有基本块都已排入流水线。流水线排布算法的伪代码如表 4-4 所示。

表 4-4

流水线排布算法

输入：attachment1.csv，dependence_L，dependence_LE，Resources
输出：L，P

```
dependence_L，dependence_LE → GW，DIN，CB，neignbors，neighbors_len
#根据依赖关系建立有向带权图 GW 和初始状态下模型参数
attachment1.csv → D #基本块所需的资源
while GW 中的基本块数量＞0：
    for CB 中的每个基本块 u：
        while(1)：
            if 把基本块 u 放入流水线 Lu 级别满足资源约束：
                break
            else：
                Lu ++ #更新基本块 u 的最早可放入级数
        end
    CB → u #从 CB 中随机选取基本块 u
    level = Lu #放入流水线 Lu 级
  Pu,level = 1
  R = D*P
  for 基本块 u 的所有后序基本块 v：
    Lv = Lu + weight(u,v)
    DINv --
        CB - {u} + {v|v 的入度为 0}
        删除 GW 的边(u,v)
  删除 GW 的删除基本块 u
```

算法的具体实现如下：

步骤 1：根据依赖关系建立有向带权图 GW 并获取模型所需参数。

1) 初始化有向带权图 GW，GW 包含所有未完成流水线排布的基本块。

①将全部基本块 B 作为 GW 的结点 GW.nodes：

$$GW.nodes = \{u|u \in B\}$$

②将基本块的依赖关系作为 GW 的边 GW.nodes：

$$GW.edges=\{(u,v)|\ \forall(u,v)\in dependence\_L\}$$
$$GW.edges=\{(u,v)|\ \forall(u,v)\in dependence\_LE\}$$

③ 根据依赖关系的不同为 GW 的边添加权重，对于基本块 u，v 的依赖关系(u,v)，如果依赖关系要求基本块 u，v 在流水线排布级数为严格小于关系，就将 GW 的边(u,v)的权重设置为 1；如果依赖关系要求基本块 u，v 在流水线排布级数小于等于，就将 GW 的边(u,v)的权重设置为 0；

$$weight_{(u,v)} = \begin{cases} 1, & (U,V) \in dependence\_L \\ 0, & (U,V) \in dependence\_LE \end{cases}$$

$$GW.edge(u,v).weight = weight_{(u,v)},\ \forall(u,v)\in dependence\_L \cup dependence\_LE$$

2) 初始化基本块入度 DIN 和 CB

① 统计 GW 中所有结点的入度：

$$DIN_u = |\{u|(v,u) \in GW.edges(), v \in B\}|,\ u \in B$$

②将入度数为 0 的点存入 CB 中，表示该点在流水线排布时间可被选择：

$$CB = \{u\ |\ u \in B\ 且 DIN_u = 0\}$$

3) 统计所有的后继结点以及后继结点的数量供后续使用：

$$neighbors_u = |\{v|(v,u) \in GW.edges(), v \in B\}|$$
$$neighbors\_len_u = |neighbors_u|$$

4) 初始化资源需求矩阵 D，流水线排布矩阵 P，最早可排入流水线 L，4 中流水线资源表 Resources

① 从 attachment1.csv 文件中读取基本块对 4 种资源的需求：

$$D_{k,u} = 基本块\ u\ 对资源\ k\ 的需求, k \in \{0,1,2,3\}, u \in B$$

② 初始状态下流水线中没有基本块：

$$P_{u,i} = 0, u \in B, i \in \{0,1,2,\dots,N\}$$

③ 初始状态下未考虑任何依赖关系，所有块的最早均可第 0 级排入流水线：

$$L_u = 0, u \in B$$

④ TCAM、HASH、ALU、QUALIFY 这 4 种资源的数量：

$$Resources = \{1,2,56,64\}$$

步骤 2：更新 GW 中所有基本块最早可放入流水线的级数。

对于基本块 u，尝试将其放入流水线 L_u 级，如果放入后满足资源资源约束，说明当

前 L_u 无需更新，如果放入后不满足资源约束，尝试放入下一级，直到找到满足资源约束的 L_u 为止。

$$L_u = \begin{cases} L_u + 1, \text{将 u 放入流水线} L_u \text{级不满足资源约束} \\ L_u, \text{将 u 放入流水线} L_u \text{级满足资源约束} \end{cases}$$

1) 尝试将基本块 u 放入流水线 Lu 级。

① 为了区分更新时的尝试放入的过程与确定放入点后的实际排布操作，将尝试放入时的流水线排布记作 P_tmp，块 u 对资源 k 的需求记作 D_tmp，块 u 尝试放入第 level 级流水线，初始化上述变量：

$$\begin{aligned} P\_tmp &= P \\ D\_tmp &= D \\ level &= L_u \end{aligned}$$

②将基本块 u 放入流水线 Lu 级：

$$P\_tmp_{u,level} = 1$$

③将尝试放入时的资源占用情况记作 R_tmp：

$$R\_tmp = D\_tmp * P\_tmp$$

2) 判断放入后是否满足资源约束 1~7。

由于放入基本块 u 前的流水线排布一定满足资源约束，放入基本块 u 后仅第 level 级流水线的资源占用情况发生了改变，因此只需要对与第 level 级流水线相关的资源约束进行判定。

①资源约束 1~4：检查第 level 级别流水线的资源占用情况超过满足最大资源，如下式所示：

$$R\_tmp_{k,level} <= resources_k, k = 0,1,2,3$$

②资源约束 5：检查与 level 级存在流水线折叠的流水线 level_fold 的资源使用是否超过 TCAM 资源的限制 1 和 HASH 资源的限制 3：

$$level\_fold = \begin{cases} level + 16, \ 0 <= level <= 15 \\ level - 16, \ 16 <= level <= 31 \\ \text{不存在流水线折叠}, \ level > 32 \end{cases}$$

$$\begin{cases} R\_tmp_{0,level} + R\_tmp_{0,level\_fold} <= 1, \text{折叠流水线满足 TCAM 资源} \\ R\_tmp_{0,level} + R\_tmp_{0,level\_fold} <= 3, \text{折叠流水线满足 TCAM 资源} \end{cases}$$

③资源约束 6：由于每级流水线 TCAM 资源只能为 0 或 1，因此只需要将偶数级流水线对 TCAM 使用情况求和即为有 TCAM 资源的偶数级流水线的数量：

$$R\_tmp_0 * C <= 5$$

④资源约束 7：由于建模求解均以基本块为单位，因此一定满足。

步骤 3：随机选取 CB 中的点 u 放入流水线并更新参数。

1) 从基本块集合 CB 中随机选取基本块 u 放入流水线 Lu 级，更新流水线状态 P 和资

源占用情况 R：

$$u = RAND()$$
$$P_{u,level} = 1$$
$$R = D * P$$

2) 根据依赖关系更新基本块 u 所有后序基本块 v 的最早流水线排布级数：

$$L_v = L_u + weight(u,v), v \in neighbors_u$$

3) 更新基本块 u 基本块入度 DIN 和 CB：

基本块 u 放入流水线后，只有和 u 存在后序依赖的基本块的入度 DIN 有变化，也只有和 u 存在后续依赖的基本块有可能被放入 CB 中。

①基本块 u 的后序块入度均减少 1：

$$DIN_v = DIN_v - 1, v \in neighbors_u$$

②将基本块 u 移 CB，并将基本块 u 的后序块中无依赖关系的块放入 CB：

$$CB - \{u\} + \{v | v \in neighbors_u 且 DIN_v = 0\}$$

4) 从 GW 中删除已排布到流水线中的基本块 u 和 u 的后序依赖边：

$$GW.edges = GW.edges - \{(u,v) | v \in neighbors_u\}$$
$$GW.nodes = GW.nodes - \{u\}$$

步骤 4：如果 GW 中仍有结点，执行步骤 2。

GW 中的结点数量不为空：

$$|GW.nodes| \neq \emptyset$$

上述方案中，我们采用**随机规则 RAND** 从 CB 中选取要加入流水线中的点，随机规则利用一定的概率和统计方法，在算法执行过程中对下一步计算做出随机选择，它的随机性体现在对数据集的操作过程是随机的、对同一数据集的计算复杂度是随机的以及输出结果是随机的。但由于初始值随机设定，以及数据的分布情况，每次所得往往会有一些差异，使得该算法并不稳定。

①RAND()：随机选取基本块 u。

$$u = CB_i, i = random()\%|CB|$$

因此，我们基于提出了下面 6 种启发式规则，按照一定的策略从 CB 中选取要放入流水线的基本块 u：

②EST()：选取最早获取的具有最早流水线级别的基本块 u。

$$u\_EST = \{u | L_u <= L_v, \forall v \in CB\}$$

$$u = u\_EST_0$$

③MST()：选取最早获取的具有最多后序基本块的基本块 u。

$$u\_MST = \{u | neighbors\_len_u <= neighbors\_len_v, \forall v \in CB\}$$

$$u = u\_MST_0$$

④**EST_MST()**：选取最早获取的具有最早流水线级别且具有最多后续基本块的基本块 u。

$$u\_EST = \{u \mid L_u <= L_v, \forall v \in CB\}$$

$$u\_EST\_MST = \{u \mid neighbors\_len_u <= neighbors\_len_v, \forall v \in u\_EST\}\ \mathrm{u} =$$

$$u = u\_EST\_MST_0$$

⑤**EST_RAND()**：随机选取具有最早流水线级别的基本块 u。

$$u\_EST = \{u \mid L_u <= L_v, \forall v \in CB\}$$

$$u = u\_EST_i, i = random()\% \mid u\_EST \mid$$

⑥**MST_RAND()**：随机选取具有最多后续基本块的基本块 u。

$$u\_MST = \{u \mid neighbors\_len_u <= neighbors\_len_v, \forall v \in CB\}$$

$$u = u\_MST_i, i = random()\% \mid u\_MST \mid$$

⑦**EST_MST_RAND()**：随机选取具有最早流水线级别且具有最多后续基本块的基本块 u。

$$u\_EST = \{u \mid L_u <= L_v, \forall v \in CB\}$$

$$u\_EST\_MST = \{u \mid neighbors\_len_u <= neighbors\_len_v, \forall v \in u\_EST\}$$

$$u = u\_EST\_MST_i, i = random()\% \mid u\_EST\_MST \mid$$

以图 4-5 的基本块排布情况为例，CB={0，1，2，3}，各基本块有图中所示的最早流水线级数和后序基本块个数，蓝色结点表示可供选择的基本块，对于 EST()、MST()、EST_MST()规则，每次可供选择的只有一块，而对于对应的随机策略，可供选的基本块有多块，且添加更多的规则，可供选择的基本块的数量越少。这样对 RAND()规则的随机性进行限制，可能会导致永远无法随机到最优状态，但是可以将最坏情况控制在一个可接受范围内。

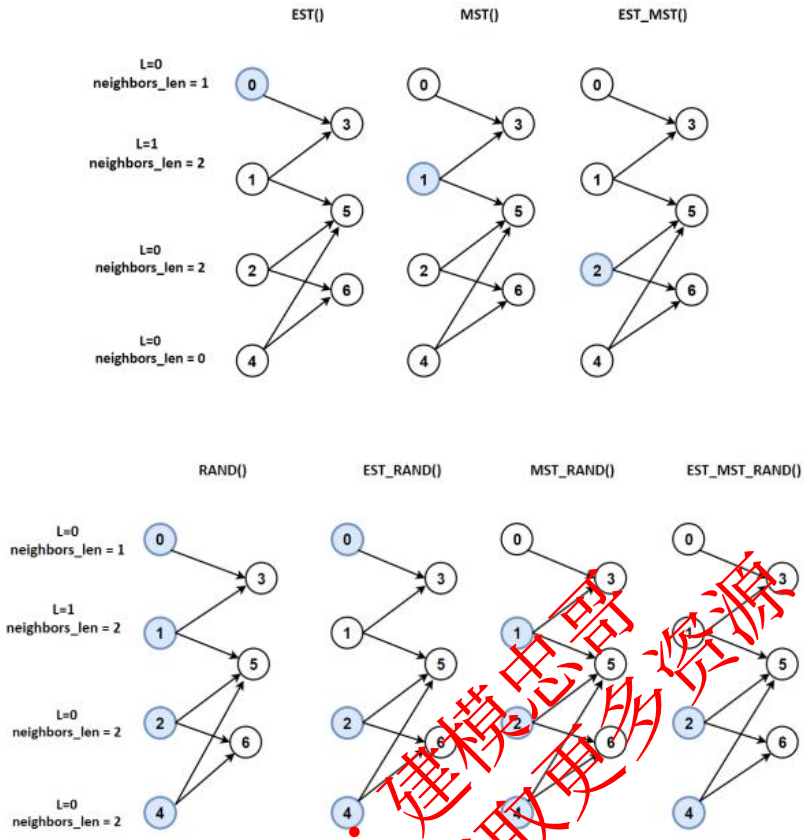

图 4-5

4.3 结果分析

（1）将附件 attachment3.csv 数据分析所得基本块间控制依赖关系可视化，如图 4-6 所示。

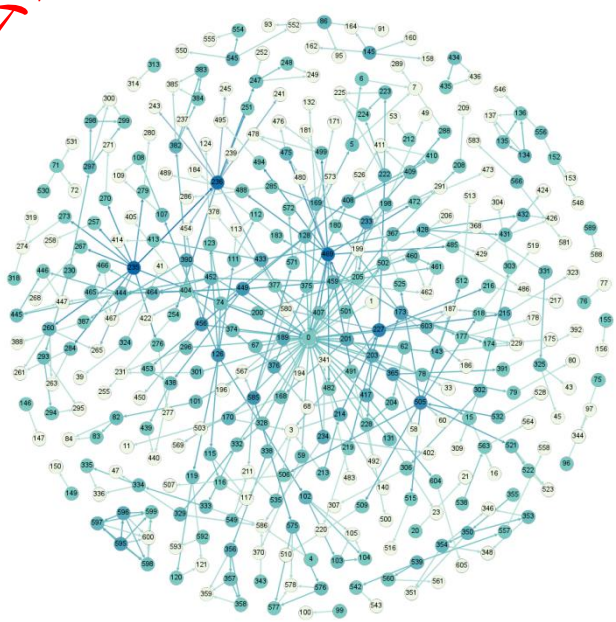

图 4-6 基本块间控制依赖关系

（2）将附件 attachment2.csv 中基本块的读写关系转化为数据依赖关系，从而确定在流水线中是严格小于还是小于等于关系的。如图 4-7 与图 4-8 所示。

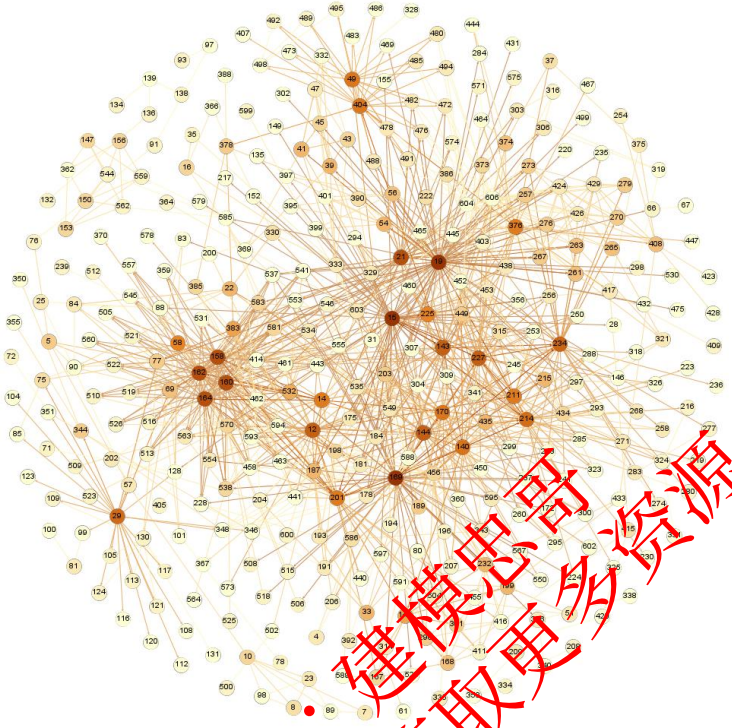

图 4-7 基本块间数据依赖小于关系

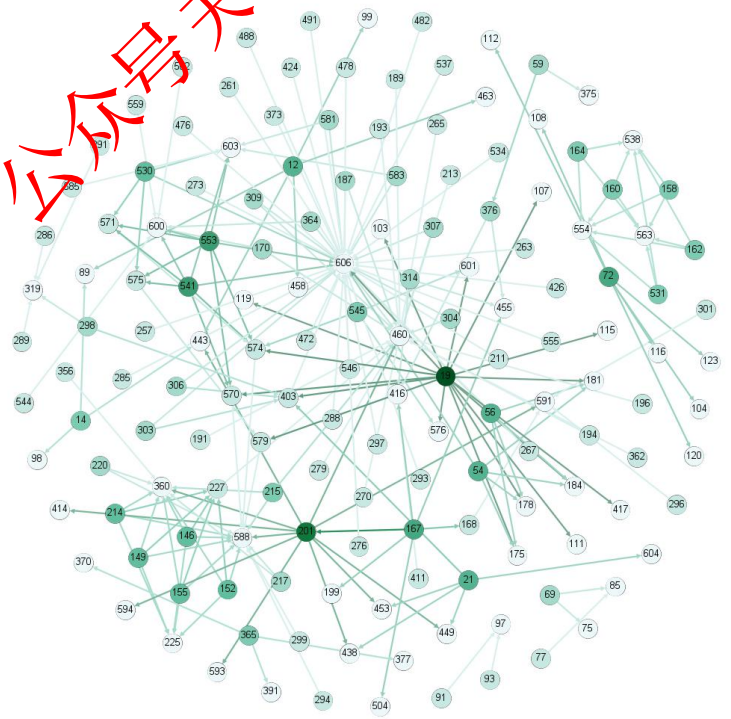

图 4-8 基本块间数据依赖小于等于关系

（3）将基本块间的控制依赖与数据依赖关系的小于等于集合取并集，可视化结果如图 4-9。

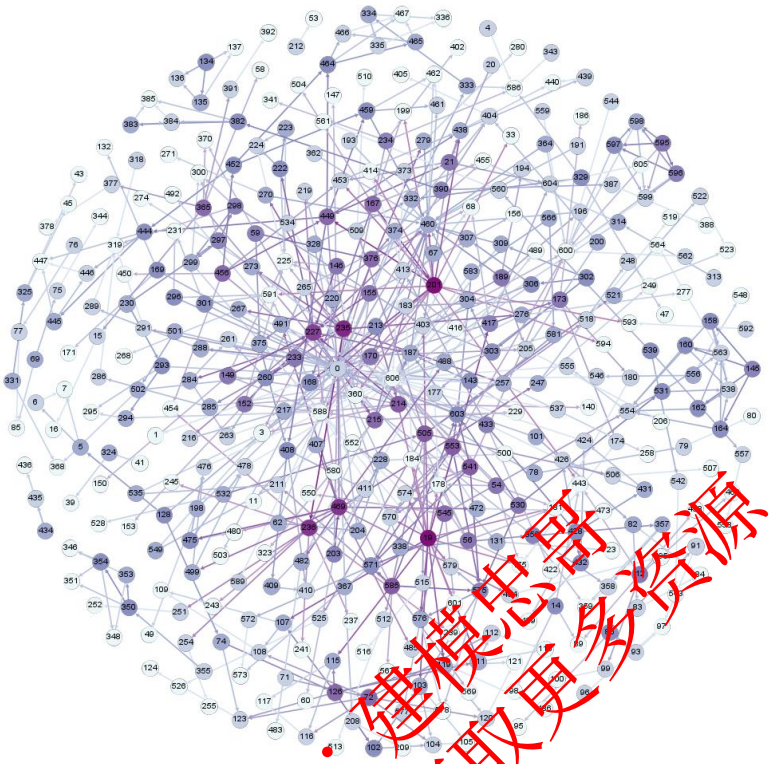

图 4-9 基本块间依赖关系

（4）表 4-5 为七种不同算法下，得到的流水线级数结果。可以看出随机算法 RAND 与随机最早开始时间算法 EST_RAND 求得的流水线占用的级数均为局部最优 40 级。但考虑到随机算法模型的不稳定性，因此选用 EST_RAND 算法更好。

表 4-5

资源排布算法	最优流水线级数
RAND	40
EST	41
MST	41
EST_RAND	40
MST_RAND	41
EST_MST	41
EST_MST_RAND	41

5 问题二的模型建立与求解

5.1 问题分析

问题二在问题一的基础上引入不在一条执行流程上的基本块的概念，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。由此，问题二对于问题一缓解了 HASH 和 ALU 的占用。问题同样要求给出基本块在流水线中的排布算法与结果，但对约束条件进行了更改。

结合分析，问题二可拆解为以下**两个子问题**：

子问题一：根据程序流程图，建立模型，确定一个基本块出发是否可到达另一个基本块，即两个基本块是否在一条执行流程上。

子问题二：结合不在同一执行流程上的基本块可共享 HASH 和 ALU 资源这一条件与更改后的约束，建立模型，确定基本块在流水线中的级数排布情况，并使占用的流水线级数尽量短。

5.2 模型建立与求解

5.2.1 模型建立

根据题目要求，在问题一基础上更改 HASH 资源约束、ALU 资源约束与折叠约束要求，并设 $E=(e_{u,v})_{N\times N}$ 为执行流程约束矩阵，基本块 $u、v \in B$，其中：

$$e_{u,v}=\begin{cases}1, & \text{基本块 u 和 v 在同一执行流程}\\ 0, & \text{基本块 u 和 v 不在同一执行流程}\end{cases}$$

建立问题二目标规划模型如下：

$$min\{\sum_{i=0}^{N} m_i\} \quad (1)$$

$$\text{s.t.}\begin{cases} L_u < L_v,\ q_{u,v}=1 & (2)\\ L_u \le L_v,\ q_{u,v}=0 & (3)\\ r_{0,i} \le 1 & (4)\\ \sum(d_{i1}+\ldots+d_{j1}) \le 2,\ L_i=\ldots=L_j \text{ and } e_{i,j}=1\ \forall i,j & (13)\\ \sum(d_{i2}+\ldots+d_{j2}) \le 56,\ L_i=\ldots=L_j \text{ and } e_{i,j}=1\ \forall i,j & (14)\\ r_{3,i} \le 64 & (7)\\ r_{0,i}+r_{0,i+16} \le 1,\ i=0\ldots15 & (8)\\ \sum(d_{i1}+\ldots+d_{j1})+\sum(d_{k1}+\ldots+d_{q1}) \le 3,\ L_i \ne L_k & (15)\\ \sum_{i=0}^{N}(r_{0,i}*C) \le 5 & (10)\\ \forall i,\ \sum_{i=0}^{N} p_{ui}=1 & (11)\end{cases}$$

其中，目标函数（1）对应问题一中的优化目标，（2）（3）（4）（7）（8）（10）（11）分别对应问题一中的约束条件，而（13）（14）（15）为问题二更改后的约束条件，分别表示了每级流水线中同一条执行流程上的基本块的 HASH 资源之和最大为 2、ALU 资源

之和最大为 56，折叠的两级对于 TCAM 资源约束不变、对于 HASH 资源，分别计算每级同一条执行流程上的基本块所占用的 HASH 资源，相加结果不超过 3。

5.2.2 模型求解

问题 2 在问题 1 的基础上增加了约束，因此需要对问题 1 的求解步骤 2-1，3-1 作出修改，其余求解步骤保持不变。步骤 2、3 具体如下：

步骤 2：更新 GW 中所有基本块最早可放入流水线的级数。

对于基本块 u，尝试将其放入流水线 L_u 级，如果放入后满足资源资源约束，说明当前 L_u 无需更新，如果放入后不满足资源约束，尝试放入下一级，直到找到满足资源约束的 L_u 为止。

$$L_u = \begin{cases} L_u + 1, \text{将 u 放入流水线} L_u \text{级不满足资源约束} \\ L_u, \text{将 u 放入流水线} L_u \text{级满足资源约束} \end{cases}$$

1）尝试将基本块 u 放入流水线 L_u 级。

① 为了区分更新时的尝试放入的过程与确定放入点后的实际排布操作，将尝试放入时的流水线排布记作 P_tmp，块 u 对资源 k 的需求记作 D_tmp，块 u 尝试放入第 level 级流水线，初始化上述变量：

$$\begin{aligned} P\_tmp &= P \\ D\_tmp &= D \\ level &= L_u \end{aligned}$$

② 将基本块 u 放入流水线 L_u 级：

$$P\_tmp_{u,level} = 1$$

③ 寻找此时也存在于流水线 level 级，且与 u 基本块不在同一执行流程上的基本块 v，由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，因此可挑选占用 HASH 资源和 ALU 资源最大的基本块，将其余基本块视为不占用本级 HASH 和 ALU 资源，将更新后的块 u 对资源 k 的需求保存于矩阵 D_tmp，更新过程伪代码如表 4-6 所示。

表 4-6

更新 D_tmp 伪代码

输入： P_tmp，level，E

输出： D_tmp

```
for 每个基本块 v:
    if P_tmp_{v,level}=1:
     if E_{u,v}=0:
        if D_tmp_{1,u} > D_tmp1_{,v}:
            D_tmp_{1,v}=0
        end
        else:
         D_tmp_{1,u}=0
        end
        if D_tmp_{2,u} > D_tmp_{2,v}:
```

11: $D\_tmp_{2,v}=0$
12: end
13: else:
14: $D\_tmp_{2,u}=0$
15: end
16: end
17: end

④ 将尝试放入时的资源占用情况记作 R_tmp：

$$R\_tmp = D\_tmp * P\_tmp$$

步骤 3：随机选取 CB 中的点 u 放入流水线并更新参数。

1）从基本块集合 CB 中随机选取基本块 u 放入流水线 L_u 级，更新流水线状态 P：

$$u = RAND()$$
$$P_{u,level} = 1$$

2) 更新基本块对资源的占用情况 D，同步骤 2，需要寻找此时存在于流水线 level 级，且与 u 基本块不在同一执行流程上的基本块 v，选择这些块中占用 HASH 资源和 ALU 资源最大的基本块，将其余可共享资源的基本块对本级 HASH 和 ALU 资源的需求记为 0。

3） 更新流水线的资源占用情况 R

$$R = D * P$$

4）根据依赖关系更新基本块 u 所有后序基本块 v 的最早流水线排布级数：

$$L_v = L_u + weight(u,v), v \in neighbors_u$$

5）更新基本块 u 基本块入度 DIN 和 CB：

①基本块 u 放入流水线后，只有和 u 存在后序依赖的基本块的入度 DIN 有变化，也只有和 u 存在后续依赖的基本块有可能被放入 CB 中。

②基本块 u 的后序块入度均减少 1：

$$DIN_v = DIN_v - 1, v \in neighbors_u$$

③将基本块 u 移 CB，并将基本块 u 的后序块中无依赖关系的块放入 CB：

$$CB - \{u\} + \{v | v \in neighbors_u 且 DIN_v = 0\}$$

6) 从 GW 中删除已排布到流水线中的基本块 u 和 u 的后序依赖边：

GW.edges = GW.edges - {(u,v)|v∈neighborsu}
GW.nodes = GW.nodes - {u}

5.3 结果分析

表 4-7 是在新约束条件下，七种不同算法模型计算出的流水线级数，易看出，各算法求得的最少流水线级数占用情况均相同，为 34 级。

表 4-7

资源排布算法	最优流水线级数
RAND	34
EST	34
MST	34
EST_RAND	34
MST_RAND	34
EST_MST	34
EST_MST_RAND	34

6 模型评价与展望

6.1 模型优点

(1) 本文针对 PISA 架构芯片资源排布问题进行研究，成功建立了随机模型 RAND 和启发式算法优化模型，并利用实验数据进行资源排布分析。求解的结果表明，本文建立的模型能够有效地解决 PISA 架构芯片资源排布问题，并具有很强的稳定性。

(2) 建立的芯片资源排布模型可解释性强，便于人们阅读理解。

(3) 基于本文构建的模型，设计启发式算法，有效降低直接求解模型带来的困难，算法难度的降低有效地增强了模型的实用性。

6.2 模型缺点

(1) 求解模型的计算量大，在高维数据下难以进行快速地求解。

(2) 模型的约束进行了理论上的简化处理，这对于模型应用到实际中有着一定的影响。

6.3 模型展望：

许多研究表明分支定价算法的效率和初始解的关系明显，本文采用的解生成方法是较简单的启发式方法，如果能够在较好的解上开始优化，那么求解效率也会提高。其次，解空间仍可以进一步缩小。可以和约束传播等方法相结合，在求解过程中进一步缩小子问题解空间，减少模型中的列，从而提高算法效率。

本文从 PISA 架构芯片资源排布问题出发，综合考虑控制依赖，数据依赖，以及各级流水线的资源约束条件，使模型有一定的应用价值，该模型的建立对于生活中其他类似调度问题也起着一定的借鉴作用。

由于能力有限，本文的模型有所不足，但只愿能为中国的芯片事业贡献一份微薄的力量。

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] Hwang C , Hsu Y , Lin Y L . Scheduling for functional pipelining and loop winding. 1991.
[4] Ferrante J, Ottenstein K J, Warren J D. The program dependence graph and its use in optimization[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1987, 9(3): 319-349.
[5] Sinha S, Harrold M J, Rothermel G. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow[C]//Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002). IEEE, 1999: 432-441.
[6] Tip F. A survey of program slicing techniques[M]. Amsterdam: Centrum voor Wiskunde en Informatica, 1994.
[7] 王军强,张松飞,陈剑,张映锋,孙树栋.一种求解资源受限多项目调度问题的分解算法[J].计算机集成制造系统,2013,19(1):83-96.
[8] 郭冬芬,李铁克.基于约束满足的车间调度算法综述[J].计算机集成制造系统,2007,13(1):117-125.

附录

附录一：控制依赖处理程序

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import csv
import scipy as sp

# 入度
inGF = []
# 第一步：dfs
dfn = []
rak = []
fa = []
search_path = []
# 第二步：sdom
val = []
bel = []
sdom = []
# 第三步：idom
idom = []
# 比较获取 CDG
cdg = []

idomGF = nx.DiGraph()


def read_csv3(file_name):
    f = open(file_name, encoding='UTF8')
    csv_reader = csv.reader(f)
    control_edges = []
    for line in csv_reader:
        adj_num = len(line)
        if adj_num > 1:
            for i in range(1, adj_num):
                control_edges.append((int(line[0]), int(line[i])))
    f.close()
    # print(control_edges)
    return control_edges


def subgraph(pointList, linkList):
    G = nx.DiGraph()
    GF = nx.DiGraph()
    # 转化为图结构
    for node in pointList:
        G.add_node(node)
```

```
        GF.add_node(node)
    for link in linkList:
        G.add_edge(link[0], link[1])
        GF.add_edge(link[1], link[0])

    return G, GF

def dfs(GF):
    # GF 的 root 为人为添加的序号最大的根
    root = len(GF.nodes) - 1
    T = nx.dfs_tree(GF, root)
    for n in GF.nodes():
        fa.append(0)
        dfn.append(n)
    global rak
    rak = list(T)  # 所有节点
    for i in range(0, len(fa)):
        dfn[rak[i]] = i
    for i in list(T.edges):  # 所有边
        fa[i[1]] = i[0]
    # print(dfn)
    # print(rak)
    # print(fa)


def find(v):
    # 还未被遍历
    if v == bel[v]:
        return v
    tmp = find(bel[v])
    if (dfn[sdom[val[bel[v]]]] < dfn[sdom[val[v]]]):
        val[v] = val[bel[v]]
    bel[v] = tmp
    return bel[v]


def tarjanFDT():
    # 初始化 val 和 sdom
    for i in range(0, len(dfn)):
        val.append(i)
        sdom.append(i)
        bel.append(i)
        idom.append(i)
    # 从大到小遍历 dfs 树
    for i in range(len(dfn) - 1, 0, -1):
        # dfs 序最大的点 u
        u = rak[i]
        # 获取 GF 原图中所有 u 的前驱
        neighbors = G.neighbors(u)
```

```
        for v in neighbors:
            find(v)
            if (dfn[sdom[val[v]]] < dfn[sdom[u]]):
                sdom[u] = sdom[val[v]]
        # print(sdom)
        sdomGF = nx.DiGraph()
        for i in range(0, len(sdom)):
            sdomGF.add_node(i)
            sdomGF.add_edge(sdom[i], i)
        bel[u] = fa[u]
        u = fa[u]
        neighbors = sdomGF.neighbors(u)
        for v in neighbors:
            find(v)
            if sdom[val[v]] == u:
                idom[v] = u
            else:
                idom[v] = val[v]

    for i in range(0, len(dfn)):
        u = rak[i]
        if idom[u] != sdom[u]:
            idom[u] = idom[idom[u]]

    global idomGF
    for i in range(0, len(idom)):
        idomGF.add_node(i)
        idomGF.add_edge(idom[i], i)

    # nx.draw_networkx(sdomGF, with_labels=True)
    # plt.show()
    # nx.draw_networkx(idomGF, with_labels=True)
    # plt.show()


def getInGF():
    # 遍历 GF 所有点
    for i in range(0, len(GF.nodes)):
        # 初始化：0 标识入度为 0
        inGF.append(0)
    for edge in GF.edges:
        inGF[edge[1]] = 1


def addGFRoot():
    # print(len(GF.nodes))
    GF.add_node(len(GF.nodes))
    for i in range(0, len(inGF)):
        if inGF[i] == 0:
            GF.add_edge(len(GF.nodes) - 1, i)
```

```
    # nx.draw_networkx(GF, with_labels=True)
    # plt.show()



if __name__ == '__main__':
    # 读 attachment3.cvs
    linkList = read_csv3("./data/attachment3.csv")
    pointList = []
    for i in range(0, 607):
        pointList.append(i)

    # 原始有向无环图 G，反向图 GF
    G, GF = subgraph(pointList, linkList)

    # 获取 GF 入度为 0 的所有点
    getInGF()

    # 为 GF 添加根节点
    addGFRoot()

    # 获取 G 的前向支配树，也就是 GF 的支配树，存储在 idomGF 中（即 FDT）
    dfs(GF)
    tarjanFDT()

    # 对比 G 原图和 FDT 图，寻找在 G 但不在 FDT 中的边，得到 CDG
    for edgeG in G.edges:
        edgeGf = (edgeG[1], edgeG[0])
        # 标识是都存在相同，初始化为 0
        flag = 0
        for edgefdt in idomGF.edges:
            if edgeGf == edgefdt:
                flag = 1
                break
            else:
                continue
        if flag == 0:
            cdg.append(edgeG)

    f = open("./data/control_dependance_less_equal.txt", "w")
    f.writelines(str(cdg))
    f.close()
    print(len(cdg))
    print(cdg)
```

附录二：数据依赖处理程序

```
import networkx as nx
import matplotlib.pyplot as plt
import ast
import csv
import collections
import numpy as np
import pandas as pd
import xlwt

np.set_printoptions(threshold=np.inf)

N = 607
control_edges = []
wr_dict = dict()
directed_edge_graph = collections.defaultdict(set)
less_list = []
less_equal_list = []
D = np.zeros((4, N))


# 从 csv1 读取一个 N*4 的矩阵 D[i][j],表示块 j 对资源 i 的需求
def read_csv1(file_name):
    global D
    D = np.loadtxt(open("./data/attachment1.csv", "rb"), int, delimiter=",",
skiprows=0)
    # print(my_matrix[4][2])
    # print(type(my_matrix))


# 从 csv3 读取最原始的控制流图到 control_edges 列表中
def read_csv3(file_name):
    f = open(file_name, encoding='UTF8')
    csv_reader = csv.reader(f)
    for line in csv_reader:
        adj_num = len(line)
        if adj_num > 1:
            for i in range(1, adj_num):
                control_edges.append((int(line[0]), int(line[i])))
    f.close()
    # print(control_edges)


# 从 csv2 读取读写关系到 wr_dict 字典中，字典的键是一个元组（资源块，
目标寄存器），值是读或写
def read_csv2(file_name):
    f = open(file_name, encoding='UTF8')
```

```
    csv_reader = csv.reader(f)
    wr_start_end = []
    wr_value = []

    for line in csv_reader:
        adj_num = len(line)
        if adj_num > 2:
            for i in range(2, adj_num):
                wr_start_end.append((line[0], line[i]))
                wr_value.append(line[1])
    f.close()
    # print(wr_start_end)
    # print(wr_value)
    # 将 2 个列表数据组合成字典
    global wr_dict
    wr_dict = dict(zip(wr_start_end, wr_value))
    # print(wr_dict)


# 有向图的两个点是否可达
def is_reachable(graph, start, end):
    dic = collections.defaultdict(list)
    #   建立图
    for u, v in graph:
        dic[u].append(v)
    #   记录访问过的节点
    vis = set()
    q = collections.deque()
    q.append(start)
    #   BFS
    while q:
        pos = q.popleft()
        if pos == end:
            return True
        for nxt in dic[pos]:
            if nxt not in vis:
                vis.add(nxt)
                q.append(nxt)
    return False

# 获取数据依赖，less_list 列表表示小于关系，less_equal_list 列表表示小于等于关系
def get_ddg():
    # 获取对每个变量，基本块的读写状态,存储在 ddg_dict 字典中，字典的键是变量，值是一个元组（基本块，读写操作）
    keys = [reg for reg in wr_dict.keys()]
    keys = list(set(keys))
    starts, ends = map(list, zip(*keys))
    ddg_dict = {}
```

```
    for reg in ends:
        ddg_dict[reg] = []
        for start_end, wr in wr_dict.items():
            if start_end[1] == reg:
                ddg_dict[reg].append((start_end[0], wr))
    print("字典键值数目：", len(ddg_dict))
    # print(ddg_dict)

    # 分别对每个变量进行数据依赖分析
    global less_list
    global less_equal_list
    # print(control_edges)
    # temp_control_edges = [(0, 1), (0, 2), (1, 2)]
    # temp_ddg_dict = {'X1': [('0', 'R'), ('1', 'W'), ('3', 'R'), ('2', 'W')], 'X2': [('0',
'W')]}
    for reg, start_wr_list in ddg_dict.items():
        if len(start_wr_list) > 1:
            for i in range(0, len(start_wr_list)):
                for j in range(i + 1, len(start_wr_list)):
                    # if start_wr_list[i][0] == '21' and start_wr_list[j][0] == '449':
                    #     print(reg, start_wr_list[i], start_wr_list[j])
                    #                 print(start_wr_list[i][0],              start_wr_list[j][0],
is_reachable(control_edges, start_wr_list[i][0], start_wr_list[j][0]))
                    # 如果存在通路，才可能有数据依赖
                    if          is_reachable(control_edges,          int(start_wr_list[i][0]),
int(start_wr_list[j][0])) != 0:
                        # 写后写，小于
                        if start_wr_list[i][1] == 'W' and start_wr_list[j][1] == 'W':
                            less_list.append((int(start_wr_list[i][0]), int(start_wr_list[j][0])))
                        # 写后读，小于
                        elif start_wr_list[i][1] == 'W' and start_wr_list[j][1] == 'R':
                            less_list.append((int(start_wr_list[i][0]), int(start_wr_list[j][0])))
                        # 读后写，小于等于
                        elif start_wr_list[i][1] == 'R' and start_wr_list[j][1] == 'W':
                            less_equal_list.append((int(start_wr_list[i][0]),
int(start_wr_list[j][0])))

    less_list = list(set(less_list))
    less_equal_list = list(set(less_equal_list))
    # print(len(less_list))
    # print(len(less_equal_list))
    # print(less_list)
    # print(less_equal_list)
    f = open("./data/data_dependance_less.txt", "w")
    f.writelines(str(less_list))
    f.close()
    f = open("./data/data_dependance_less_equal.txt", "w")
    f.writelines(str(less_equal_list))
    f.close()
```

```
# 合并两个小于等于的依赖
def get_all_dependance():
    with open('./data/data_dependance_less_equal.txt', 'r') as f1:
        list1 = []
        for line in f1:
            list1 = ast.literal_eval(line)
    with open('./data/control_dependance_less_equal.txt', 'r') as f2:
        list2 = []
        for line in f2:
            list2 = ast.literal_eval(line)

    all_less_equal = list(set(list1).union(set(list2)))
    f = open("./data/all_less_equal.txt", "w")
    f.writelines(str(all_less_equal))
    f.close()

# 相同流程上的点集合
def get_same_flow(E):
    for i in range(0, N):
        for j in range(0, N):
            if E[i][j] != 1:
                if is_reachable(control_edges, i, j) != 0:
                    E[i][j] = 1
                    E[j][i] = 1

    # f = open("./data/same_flow_matrix.txt", "w")
    # f.writelines(str(E))
    df = pd.DataFrame(E)
    df.to_csv('./data/same_flow_matrix.csv', index=False, header=False)


if __name__ == '__main__':
    read_csv3("./data/attachment3.csv")
    read_csv2("./data/attachment2.csv")
    read_csv1("./data/attachment1.csv")
    get_ddg()
    E = np.zeros((N, N))
    get_all_dependance()
    get_same_flow(E)
    # print(E)
```


附录三：问题一求解程序

```
import collections

import networkx as nx
import matplotlib.pyplot as plt
```

```
import numpy as np
import random
import copy
import ast
import sys
import os

np.set_printoptions(threshold=np.inf)

N = 607
# 带权有向无环图
GW = nx.DiGraph()
# D[i][j] 块 j 对资源 i 的需求
D = np.zeros((4, N))
# P[i][j] 当前流水线排布状态：块 i 放置在流水线 j 级
P = np.zeros((N, N))
# R[i][j] 当前资源使用情况，第 i 级流水线对资源 j 的使用量
R = np.zeros((4, N))
# L[i] 块 i 可放入的最早流水线级数，i=0,...,N
L = [0] * N
# DIN[i] 当前第 i 块的入度
DIN = [0] * N
# 当前入读为 0 块的列表
ListZeroDin = []
# resources[k] 提供的资源 k
resources = [1, 2, 56, 64]
# 偶数级数组

even_num = np.zeros((607, 1))

# 小于依赖关系 小于等于依赖关系表
dependence_L = []
dependence_LE = []

edge_labels = {}

# 后继结点字典
neighbors = {}
# 后继结点数量
neighbors_len = [0] * N

# random_test = []


def initial():
    global P, L
    P = np.zeros((N, N))
    L = [0] * N
```

```
    # initial_even：为后续偶数级约束 6 做准备
    initial_even()
    # initial_GW：初始化 GW，添加节点与边，权值 0 表示小于等于，权值 1
表示小于
    initial_GW()
    # initial_D：读 attachment1.csv 可能需要读后转置
    initial_D()
    # initial_DIN：统计 GW 所有结点入度
    initial_DIN()
    # get_ListZeroDin：获取所有入读为 0 的结点
    initial_ListZeroDin()
    # initial_neighbors：初始化 GW 的后继结点以及每个结点后继结点数量
    initial_neighbors()
    return


def initial_neighbors():
    for node in GW.nodes:
        tmp_n = []
        for i in GW.neighbors(node):
            tmp_n.append(i)
        neighbors[node] = tmp_n
        neighbors_len[node] = len(tmp_n)
    # print(neighbors)
    # print(neighbors_len)


# 有向图的两个点是否可达
def is_reachable(graph, start, end):
    dic = collections.defaultdict(list)
    #  建立图
    for u, v in graph:
        dic[u].append(v)
    #  记录访问过的节点
    vis = set()
    q = collections.deque()
    q.append(start)
    #  BFS
    while q:
        pos = q.popleft()
        if pos == end:
            return True
        for nxt in dic[pos]:
            if nxt not in vis:
                vis.add(nxt)
                q.append(nxt)
    return False
```

```
def initial_even():
    global even_num
    for i in range(0, N):
        if i % 2 == 0:
            even_num[i][0] = 1
        else:
            even_num[i][0] = 0


def initial_DIN():
    for edge in GW.edges:
        DIN[edge[1]] += 1


def initial_D():
    global D
    # D = [[0,1,0,1],
    #      [0,0,0,0],
    #      [2,0,0,0],
    #      [0,0,0,0]]

    D = np.loadtxt(open("./data/attachment\1.csv", "rb"), int, delimiter=",", skiprows=0)
    D = D.T


def initial_GW():
    # TODO:读依赖关系
    # dependence_LE = [(0, 2)]
    # dependence_L = [(0, 1)]

    with open('./data/data_dependance_less.txt', 'r') as f1:
        for line in f1:
            dependence_L = ast.literal_eval(line)
    with open('./data/all_less_equal.txt', 'r') as f2:
        for line in f2:
            dependence_LE = ast.literal_eval(line)

    for i in range(0, N):
        GW.add_node(i)
    for edge in dependence_LE:
        GW.add_edge(edge[0], edge[1], name=0)
    for edge in dependence_L:
        GW.add_edge(edge[0], edge[1], name=1)

    global edge_labels
    edge_labels = nx.get_edge_attributes(GW, 'name')


def initial_ListZeroDin():
```

```
    for node in range(0, N):
        if DIN[node] == 0:
            ListZeroDin.append(node)
    # print(ListZeroDin)


# EST（最早开始时间）：选取可排布块中最早流水线级别块
def select_EST():
    tmp = N
    node = 0
    for i in ListZeroDin:
        if L[i] < tmp:
            tmp = L[i]
            node = i
        else:
            continue
    return node


def select_EST_RAND():
    tmp = N
    tmp_node = []
    for i in ListZeroDin:
        if L[i] < tmp:
            # 清空旧 list 构建新的
            tmp_node.clear()
            tmp = L[i]
            tmp_node.append(i)
        if L[i] == tmp:
            tmp_node.append(i)
        else:
            continue
    index = random.randint(0, len(tmp_node) - 1)
    node = tmp_node[index]
    return node


# MST（最多紧后工序）：选择可排布块中后继结点数最多的块
def select_MST():
    tmp = -1
    node = 0
    for i in ListZeroDin:
        if neighbors_len[i] > tmp:
            tmp = neighbors_len[i]
            node = i
        else:
            continue
    return node
```

```
def select_MST_RAND():
    tmp = -1
    tmp_node = []
    for i in ListZeroDin:
        if neighbors_len[i] > tmp:
            tmp_node.clear()
            tmp = neighbors_len[i]
            tmp_node.append(i)
        if neighbors_len[i] == tmp:
            tmp_node.append(i)
        else:
            continue
    index = random.randint(0, len(tmp_node) - 1)
    node = tmp_node[index]
    return node


# RAND（随机生成方式）：随机选择可排布块中的块
def select_RAND():
    # random_test = np.random.randint(0, len(ListZeroDin), 1)  # 前闭后开
    index = random.randint(0, len(ListZeroDin) - 1)  # 前闭后闭
    node = ListZeroDin[index]
    # node = ListZeroDin[random_test[0]]
    return node


# EST_MST(最早开始+最多后继)
def select_EST_MST():
    tmp = N
    tmp_node_est = []
    node = 0
    # 所有最早存入 tmp_node
    for i in ListZeroDin:
        if L[i] < tmp:
            # 清空旧 list 构建新的
            tmp_node_est.clear()
            tmp = L[i]
            tmp_node_est.append(i)
        if L[i] == tmp:
            tmp_node_est.append(i)
        else:
            continue
    # 最早里面选最多后继
    tmp = -1
    for i in tmp_node_est:
        if neighbors_len[i] > tmp:
            tmp = neighbors_len[i]
            node = i
        else:
```

```
            continue
    return node


# EST_MST_RAND(最早开始+最多后继+随机)
def select_EST_MST_RAND():
    tmp = N
    tmp_node_est = []
    # 所有最早存入 tmp_node
    for i in ListZeroDin:
        if L[i] < tmp:
            # 清空旧 list 构建新的
            tmp_node_est.clear()
            tmp = L[i]
            tmp_node_est.append(i)
        if L[i] == tmp:
            tmp_node_est.append(i)
        else:
            continue
    # 最早里面选最多后继
    tmp = -1
    tmp_node_est_mst = []
    for i in tmp_node_est:
        if neighbors_len[i] > tmp:
            tmp_node_est_mst.clear()
            tmp = neighbors_len[i]
            tmp_node_est_mst.append(i)
        if neighbors_len[i] == tmp:
            tmp_node_est_mst.append(i)
        else:
            continue
    index = random.randint(0, len(tmp_node_est_mst) - 1)
    node = tmp_node_est_mst[index]
    return node


# MST_EST(最多后继+最早开始)
def select_MST_EST():
    tmp = -1
    tmp_node_mst = []
    for i in ListZeroDin:
        if neighbors_len[i] > tmp:
            tmp_node_mst.clear()
            tmp = neighbors_len[i]
            tmp_node_mst.append(i)
        if neighbors_len[i] == tmp:
            tmp_node_mst.append(i)
        else:
            continue
```

```
    tmp = N
    node = 0
    for i in tmp_node_mst:
        if L[i] < tmp:
            tmp = L[i]
            node = i
        else:
            continue
    return node


# MST_EST_RAND(最早后继+最早开始)
def select_MST_EST_RAND():
    tmp = -1
    tmp_node_mst = []
    for i in ListZeroDin:
        if neighbors_len[i] > tmp:
            tmp_node_mst.clear()
            tmp = neighbors_len[i]
            tmp_node_mst.append(i)
        if neighbors_len[i] == tmp:
            tmp_node_mst.append(i)
        else:
            continue
    tmp = N
    tmp_node_mst_est = []
    for i in tmp_node_mst:
        if L[i] < tmp:
            # 清空旧 list 构建新的
            tmp_node_mst_est.clear()
            tmp = L[i]
            tmp_node_mst_est.append(i)
        if L[i] == tmp:
            tmp_node_mst_est.append(i)
        else:
            continue
    index = random.randint(0, len(tmp_node_mst_est) - 1)
    node = tmp_node_mst_est[index]
    return node


def meet_ResourceCons(node):
    P_tmp = copy.deepcopy(P)
    P_tmp[node][L[node]] = 1
    R_tmp = np.dot(D, P_tmp)
    level = L[node]

    # 约束(1)(2)(3)(4)：放入第 level 级流水线，判断该级的四种资源
    for k in range(0, len(resources)):
        if R_tmp[k][level] <= resources[k]:
```

```
            continue
        else:
            return False
    # 约束(5):仅在结点数不少于 16 时才需要约束
    if (N >= 16):
        if level >= 0 & level < 16:
            if R_tmp[0][level] + R_tmp[0][level + 16] > 1:
                return False
            if R_tmp[1][level] + R_tmp[1][level + 16] > 3:
                return False
        if level >= 16 & level < 32:
            if R_tmp[0][level] + R_tmp[0][level - 16] > 1:
                return False
            if R_tmp[1][level] + R_tmp[1][level - 16] > 3:
                return False
    # print("meet (5)")
    # 约束(6)
    if np.dot(R_tmp[0], even_num) > 5:
        return False
    # print("meet (6)")
    return True


# 依据资源约束更新块 i 的最早流水线级数 L（resource）
def update_LR(node):
    while (1):
        # Meet resource constraints
        if meet_ResourceCons(node):
            # print(node, "满足资源要求")
            break
        else:
            # print(node, "所需资源：", D.T[node])
            # print("当前各级流水线资源占用情况：", R)
            # print("目前的流水线分布情况：", P)
            L[node] += 1


# 依据控制和数据依赖关系更新所有后继结点的最先流水线级数（dependence）,L[v]=L[u]+weight(u,v)
def update_LD(node):
    for neighbor in GW.neighbors(node):
        tup = (node, neighbor)
        # print(edge_labels.get(tup))
        L[neighbor] = L[node] + edge_labels.get(tup)


def update_GW(node):
    neighborList = []
    for neighbor in GW.neighbors(node):
```

```
        neighborList.append(neighbor)
    for neighbor in neighborList:
        GW.remove_edge(node, neighbor)
    # print(node)
    # print(neighborList)
    GW.remove_node(node)


def update_DIN(node):
    # print("update_DIN")
    for neighbor in GW.neighbors(node):
        DIN[neighbor] -= 1
    # print("DIN: " ,DIN)


def update_ListZeroDin(node):
    ListZeroDin.remove(node)
    for neighbor in GW.neighbors(node):
        if DIN[neighbor] == 0:
            ListZeroDin.append(neighbor)
    # print("入度为 0 的点: ", ListZeroDin)


def put_Block(node):
    # print("更新的节点为：", node)
    # 更新 P 流水线状态，把 node 加入流水线，更新当前资源使用情况
    P[node][L[node]] = 1
    global R
    R = np.dot(D, P)
    # print("len(GW.nodes):", len(GW.nodes))
    # 依据控制和数据依赖关系更新所有后继结点的最先流水线级数（dependence）,L[v]=L[u]+weight(u,v)
    update_LD(node)
    # 更新所有后继节点的入度数以及 ListZeroDin
    update_DIN(node)
    update_ListZeroDin(node)
    # 更新 GW,删除 node 以及 node 出发的所有边
    update_GW(node)
    # print("当前节点", node, "成功加入流水线")


if __name__ == '__main__':
    fw = open("./log/question1_EST_MST_RAND.txt", 'a')
    for i in range(0, 9):
        initial()
        # 逐个将结点加入流水线，加入后从 GW 删除结点和边
        while len(GW.nodes) > 0:
            # 更新所有可排布结点（入度为 0）的最早流水线级数
            # print(ListZeroDin) [0,3]
```

```
            for node in ListZeroDin:
                # 依据资源约束更新块 i 的最早流水线级数（resource）
                update_LR(node)
            # EST（最早开始时间）：选取 ListZeroDin 中最早流水线级别块
            # node = select_EST()
            # MST（最多紧后工序）：选择 ListZeroDin 中后继结点数最多的块
            # node = select_MST()
            # RAND（随机生成方式）：随机选择 ListZeroDin 中的块
            # node = select_RAND()

            # EST_RAND（最早开始时间+随机生成方式）：随机选择 ListZeroDin 中的块
            # node = select_EST_RAND()
            # MST_RAND（最多后继+随机生成方式）：随机选择 ListZeroDin 中的块
            # node = select_MST_RAND()
            # EST_MST(最早开始+最多后继)
            # node = select_EST_MST()
            # EST_MST_RAND(最早开始+最多后继)
            # node = select_EST_MST_RAND()

            # 把 block 放入流水线 L[block]的位置
            put_Block(node)
            # print("目前的各个块所能放的最早流水线级数 L[i]:", L)


        pipeline_leval = max(L) + 1
        fw.write("流水线总数：\n")
        fw.write(str(pipeline_leval))
        fw.write("\n")

        fw.write("流水线分布情况：\n")
        fw.write(str(L))
        fw.write("\n")

        fw.write("流水线资源占用情况：\n")
        fw.write(str(R[:, 0:pipeline_leval + 1]))
        fw.write("\n")

        print("第 ", i, " 次实验流水线级数：\n", pipeline_leval)
    fw.close()
```


附录四：问题二求解程序

```
import networkx as nx
import matplotlib.pyplot as plt
```

```
import numpy as np
import random
import copy
import ast
import sys
import os

np.set_printoptions(threshold=np.inf)

N = 607
# 带权有向无环图
GW = nx.DiGraph()
# D[i][j] 块 j 对资源 i 的需求
D = np.zeros((4, N))
# P[i][j] 当前流水线排布状态：块 i 放置在流水线 j 级
P = np.zeros((N, N))
# R[i][j] 当前资源使用情况，第 j 级流水线对资源 i 的使用量
R = np.zeros((4, N))
# L[i] 块 i 可放入的最早流水线级数，i=0,...,N-1
L = [0] * N
# DIN[i] 当前第 i 块的入度
DIN = [0] * N
# 当前入读为 0 块的列表
ListZeroDin = []
# resources[k] 提供的资源 k
resources = [1, 2, 56, 64]
# 偶数级数组
even_num = np.zeros((N, 1))

# 小于依赖关系 小于等于依赖关系表
dependence_L = []
dependence_LE = []

edge_labels = {}

# 后继结点字典
neighbors = {}
# 后继结点数量
neighbors_len = [0] * N

# E[i][j] 执行流程依赖矩阵，1：i 和 j 在一条执行流程，0：i 和 j 不在一条执行流程
E = np.zeros((N, N))


def initial():
    global P, L
    P = np.zeros((N, N))
```

```
    L = [0] * N
    # initial_even：为后续偶数级约束 6 做准备
    initial_even()
    # initial_GW：初始化 GW，添加节点与边，权值 0 表示小于等于，权值 1 表示小于
    initial_GW()
    # initial_D：读 attachment1.csv 可能需要读后转置
    initial_D()
    # initial_DIN：统计 GW 所有结点入度
    initial_DIN()
    # initial_E：执行流程依赖关系
    initial_E()
    # get_ListZeroDin：获取所有入读为 0 的结点
    initial_ListZeroDin()
    # initial_neighbors：初始化 GW 的后继结点以及每个结点后继结点数量
    initial_neighbors()
    return


def initial_neighbors():
    for node in GW.nodes:
        tmp_n = []
        for i in GW.neighbors(node):
            tmp_n.append(i)
        neighbors[node] = tmp_n
        neighbors_len[node] = len(tmp_n)
    # print(neighbors)
    # print(neighbors_len)


def initial_E():
    global E
    E = np.loadtxt(open("./data/same_flow_matrix.csv"), delimiter=",", skiprows=0)
    # print(E)


def initial_even():
    global even_num
    for i in range(0, N):
        if i % 2 == 0:
            even_num[i][0] = 1
        else:
            even_num[i][0] = 0


def initial_DIN():
    for edge in GW.edges:
        DIN[edge[1]] += 1
```

```
def initial_D():
    global D
    D = np.loadtxt(open("./data/attachment1.csv", "rb"), int, delimiter=",", skiprows=0)
    D = D.T


def initial_GW():
    dependence_LE = [(0, 2)]
    dependence_L = [(0, 1)]

    for i in range(0, N):
        GW.add_node(i)
    for edge in dependence_LE:
        GW.add_edge(edge[0], edge[1], name=0)
    for edge in dependence_L:
        GW.add_edge(edge[0], edge[1], name=1)

    global edge_labels
    edge_labels = nx.get_edge_attributes(GW, 'name')

def initial_ListZeroDin():
    for node in range(0, N):
        if DIN[node] == 0:
            ListZeroDin.append(node)
    # print(ListZeroDin)


# EST（最早开始时间）：选取可排布块中最早流水线级别块
def select_EST():
    tmp = N
    node = 0
    for i in ListZeroDin:
        if L[i] < tmp:
            tmp = L[i]
            node = i
        else:
            continue
    return node


def select_EST_RAND():
    tmp = N
    tmp_node = []
    for i in ListZeroDin:
        if L[i] < tmp:
            # 清空旧 list 构建新的
            tmp_node.clear()
            tmp = L[i]
```

```
            tmp_node.append(i)
        if L[i] == tmp:
            tmp_node.append(i)
        else:
            continue
    index = random.randint(0, len(tmp_node) - 1)
    node = tmp_node[index]
    return node


# MST（最多紧后工序）：选择可排布块中后继结点数最多的块
def select_MST():
    tmp = -1
    node = 0
    for i in ListZeroDin:
        if neighbors_len[i] > tmp:
            tmp = neighbors_len[i]
            node = i
        else:
            continue
    return node


def select_MST_RAND():
    tmp = -1
    tmp_node = []
    for i in ListZeroDin:
        if neighbors_len[i] > tmp:
            tmp_node.clear()
            tmp = neighbors_len[i]
            tmp_node.append(i)
        if neighbors_len[i] == tmp:
            tmp_node.append(i)
        else:
            continue
    index = random.randint(0, len(tmp_node) - 1)
    node = tmp_node[index]
    return node


# RAND（随机生成方式）：随机选择可排布块中的块
def select_RAND():
    index = random.randint(0, len(ListZeroDin) - 1)
    node = ListZeroDin[index]
    return node


# EST_MST(最早开始+最多后继)
def select_EST_MST():
    tmp = N
```

```
    tmp_node_est = []
    node = 0
    # 所有最早存入 tmp_node
    for i in ListZeroDin:
        if L[i] < tmp:
            # 清空旧 list 构建新的
            tmp_node_est.clear()
            tmp = L[i]
            tmp_node_est.append(i)
        if L[i] == tmp:
            tmp_node_est.append(i)
        else:
            continue
    # 最早里面选最多后继
    tmp = -1
    for i in tmp_node_est:
        if neighbors_len[i] > tmp:
            tmp = neighbors_len[i]
            node = i
        else:
            continue
    return node


# EST_MST_RAND(最早开始+最多后继+随机)
def select_EST_MST_RAND():
    tmp = N
    tmp_node_est = []
    # 所有最早存入 tmp_node
    for i in ListZeroDin:
        if L[i] < tmp:
            # 清空旧 list 构建新的
            tmp_node_est.clear()
            tmp = L[i]
            tmp_node_est.append(i)
        if L[i] == tmp:
            tmp_node_est.append(i)
        else:
            continue
    # 最早里面选最多后继
    tmp = -1
    tmp_node_est_mst = []
    for i in tmp_node_est:
        if neighbors_len[i] > tmp:
            tmp_node_est_mst.clear()
            tmp = neighbors_len[i]
            tmp_node_est_mst.append(i)
        if neighbors_len[i] == tmp:
            tmp_node_est_mst.append(i)
```

```
        else:
            continue
    index = random.randint(0, len(tmp_node_est_mst) - 1)
    node = tmp_node_est_mst[index]
    return node


# MST_EST(最多后继+最早开始)
def select_MST_EST():
    tmp = -1
    tmp_node_mst = []
    for i in ListZeroDin:
        if neighbors_len[i] > tmp:
            tmp_node_mst.clear()
            tmp = neighbors_len[i]
            tmp_node_mst.append(i)
        if neighbors_len[i] == tmp:
            tmp_node_mst.append(i)
        else:
            continue
    tmp = N
    node = 0
    for i in tmp_node_mst:
        if L[i] < tmp:
            tmp = L[i]
            node = i
        else:
            continue
    return node


# MST_EST_RAND(最早后继+最早开始)
def select_MST_EST_RAND():
    tmp = -1
    tmp_node_mst = []
    for i in ListZeroDin:
        if neighbors_len[i] > tmp:
            tmp_node_mst.clear()
            tmp = neighbors_len[i]
            tmp_node_mst.append(i)
        if neighbors_len[i] == tmp:
            tmp_node_mst.append(i)
        else:
            continue
    tmp = N
    tmp_node_mst_est = []
    for i in tmp_node_mst:
        if L[i] < tmp:
            # 清空旧 list 构建新的
            tmp_node_mst_est.clear()
```

```
            tmp = L[i]
            tmp_node_mst_est.append(i)
        if L[i] == tmp:
            tmp_node_mst_est.append(i)
        else:
            continue
    index = random.randint(0, len(tmp_node_mst_est) - 1)
    node = tmp_node_mst_est[index]
    return node


def meet_ResourceCons(node):
    P_tmp = copy.deepcopy(P)
    D_tmp = copy.deepcopy(D)
    P_tmp[node][L[node]] = 1
    level = L[node]

    # 判断与当前流水线是否存在不在统一执行流程点，如果存在，说明可以合并 D(demond)
    for i in range(0, N):
        if P_tmp[i][level]:
            if E[node][i] == 0:
                if D_tmp[1][node] > D_tmp[1][i]:
                    D_tmp[1][i] = 0
                else:
                    D_tmp[1][node] = 0
                if D_tmp[2][node] > D_tmp[2][i]:
                    D_tmp[2][i] = 0
                else:
                    D_tmp[2][node] = 0
    R_tmp = np.dot(D_tmp, P_tmp)

    # print("upadate L: node ", node, ", L is ", L[node])
    # 约束(1)(2)(3)(4)：放入第 level 级流水线，判断该级的四种资源
    for k in range(0, len(resources)):
        if R_tmp[k][level] <= resources[k]:
            continue
        else:
            return False
    # print("meet (1)(2)(3)(4)")
    # 约束(5):仅在结点数不少于 16 时才需要约束
    if (N >= 16):
        if level >= 0 & level < 16:
            if R_tmp[0][level] + R_tmp[0][level + 16] > 1:
                # print("here 0")
                return False
            if R_tmp[1][level] + R_tmp[1][level + 16] > 3:
                # print("here 1")
                return False
```

```
        if level >= 16 & level < 32:
            if R_tmp[0][level] + R_tmp[0][level - 16] > 1:
                return False
            if R_tmp[1][level] + R_tmp[1][level - 16] > 3:
                return False
    # print("meet (5)")
    # 约束(6)
    if np.dot(R_tmp[0], even_num) > 5:
        return False
    # print("meet (6)")
    return True


# 依据资源约束更新块 i 的最早流水线级数 L（resource）
def update_LR(node):
    while (1):
        # Meet resource constraints
        if meet_ResourceCons(node):
            # print(node, "满足资源要求")
            break
        else:
            # print(node, "所需资源：", D.T[node])
            # print("当前各级流水线资源占用情况：", R)
            # print("目前的流水线分布情况：", P)
            L[node] += 1


# 依据控制和数据依赖关系更新所有后继结点的最先流水线级数（dependence）,L[v]=L[u]+weight(u,v)
def update_LD(node):
    # print("update_LD")
    for neighbor in GW.neighbors(node):
        tup = (node, neighbor)
        # print(edge_labels.get(tup))
        L[neighbor] = L[node] + edge_labels.get(tup)


def update_GW(node):
    neighborList = []
    for neighbor in GW.neighbors(node):
        neighborList.append(neighbor)
    for neighbor in neighborList:
        GW.remove_edge(node, neighbor)
    # print(node)
    # print(neighborList)
    GW.remove_node(node)


def update_DIN(node):
```

```
    # print("update_DIN")
    for neighbor in GW.neighbors(node):
        DIN[neighbor] -= 1
    # print("DIN: " ,DIN)


def update_ListZeroDin(node):
    ListZeroDin.remove(node)
    for neighbor in GW.neighbors(node):
        if DIN[neighbor] == 0:
            ListZeroDin.append(neighbor)
    # print("入度为 0 的点: ", ListZeroDin)


def put_Block(node):
    # print("更新的节点为：", node)
    # 更新 P 流水线状态，把 node 加入流水线，更新当前资源使用情况
    P[node][L[node]] = 1
    global R
    level = L[node]
    for i in range(0, N):
        if P[i][level]:
            if E[node][i] == 0:
                if D[1][node] > D[1][i]:
                    D[1][i] = 0
                else:
                    D[1][node] = 0
                if D[2][node] > D[2][i]:
                    D[2][i] = 0
                else:
                    D[2][node] = 0
    R = np.dot(D, P)
    # print("len(GW.nodes):", len(GW.nodes))
    # 依据控制和数据依赖关系更新所有后继结点的最先流水线级数（dependence）,L[v]=L[u]+weight(u,v)
    update_LD(node)
    # 更新所有后继节点的入度数以及 ListZeroDin
    update_DIN(node)
    update_ListZeroDin(node)
    # 更新 GW,删除 node 以及 node 出发的所有边
    update_GW(node)
    # print("当前节点", node, "成功加入流水线")


if __name__ == '__main__':
    fw = open("./log/question2_RAND.txt", 'a')
    for i in range(0, 10):
        initial()
        # 逐个将结点加入流水线，加入后从 GW 删除结点和边
```

```
        while len(GW.nodes) > 0:
            # 更新所有可排布结点（入度为 0）的最早流水线级数
            # print(ListZeroDin) [0,3]
            for node in ListZeroDin:
                # 依据资源约束更新块 i 的最早流水线级数（resource）
                update_LR(node)
            # EST（最早开始时间）：选取 ListZeroDin 中最早流水线级别块
            # node = select_EST()
            # MST（最多紧后工序）：选择 ListZeroDin 中后继结点数最多的块
            # node = select_MST()
            # RAND（随机生成方式）：随机选择 ListZeroDin 中的块
            node = select_RAND()

            # EST_RAND（最早开始时间+随机生成方式）：随机选择 ListZeroDin 中的块
            # node = select_EST_RAND()
            # MST_RAND（最多后继+随机生成方式）：随机选择 ListZeroDin 中的块
            # node = select_MST_RAND()
            # EST_MST(最早开始+最多后继)
            # node = select_EST_MST()
            # EST_MST_RAND(最早开始+最多后继)
            # node = select_EST_MST_RAND()

            # 把 block 放入流水线 L[block]的位置
            put_Block(node)
            # print("目前的各个块所能放的最早流水线级数 L[i]:", L)

        pipeline_leval = max(L) + 1
        fw.write("流水线总数：\n")
        fw.write(str(pipeline_leval))
        fw.write("\n")

        fw.write("流水线分布情况：\n")
        fw.write(str(L))
        fw.write("\n")

        fw.write("最新资源占用情况：\n")
        fw.write(str(D))
        fw.write("\n")

        fw.write("流水线资源占用情况：\n")
        fw.write(str(R[:, 0:pipeline_leval + 1]))
        fw.write("\n")

        print("第 ", i, " 次实验流水线级数：\n", pipeline_leval)
    fw.close()
```