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	清华大学
参赛队号	22900310001
队员姓名	1. 刘兴禄
	2. 赖克凡
	3. 曹可欣

中国光谷·“华为杯”第十九届中国研究生数学建模竞赛

题 目 芯片资源排布优化的高效精确求解算法

摘要

本文研究了网络通信领域的交换芯片资源排布优化问题。基于问题的假设，我们建立了考虑资源约束、资源共享、数据依赖和控制依赖的芯片资源排布优化整数规划模型。该问题属于 NP-hard，求解难度较大。为了加速求解，我们针对 2 个问题的模型分别设计了相应的高效精确求解算法，即 **标松弛诱导定界和变量固定驱动的分支切割算法** 和 **松弛和约束生成-可行性证明定界算法**。这两种精确算法通过目标松弛快速得到可行解或者证明不可行性，以此来获得更紧的上界和下界；通过目标界限的收紧来固定变量的取值，从而缩减问题规模；通过约束的松弛显著减少约束的数量。本文提出的算法极大地缩短了求解时间，并且保证了解的全局最优性。对于问题所提供的算例，这两个算法分别将两个问题求解到了 **全局最优解**，且显著加速了求解过程 (得到两个问题的最优解的时间分别为**1.5 小时** 和**183 秒**)。具体来讲，**问题一的最优解为 68**，**问题二的最优解为 39**，即问题一的基本块排布方案只需 68 个流水级，问题二的基本块排布方案只需 39 个流水级。

问题一中除了要考虑多种资源约束之外，还需要考虑基本块之间的数据依赖和控制依赖关系。数据依赖的关系只需要构建数据依赖关系网络图，并通过判断可达性即可得出，但是控制依赖的识别较为复杂。为了准确且高效地识别控制依赖关系，我们提出了一种 **最大可达网络流模型**，并设计了一种复杂度为 $\mathcal{O}(n^2)$ 的高效的**广度前向传播算法**来求解该模型。基于广度前向传播算法的求解结果，可以准确建立出芯片资源排布整数规划模型。该模型变量个数最多为 369054，约束个数为 129830。直接调用 Gurobi 求解该模型效率非常低，运行 25 小时仍未找到任何可行解。为了加速求解，我们提出了一种精确的 **目标松弛诱导定界和变量固定驱动的分支切割算法**。该算法通过目标松弛方法，仅用 **35 分钟** 左右就获得了下界 66 和上界 71。通过变量固定步骤，我们 **固定了 98% 的决策变量**，极大地缩减了问题规模。通过进一步的探索，算法最终在运行 **1.5 小时** 后得到了 **目标值为 68 的全局最优解 (Gap=0)**。结果显示，在最优的排布方案中，四种资源的平均利用率达到了 71.5%，平均每一级分配 9 个基本块。第 7 级被分配的基本块最多，为 34 个；第 9 级和第 35 级被分配的基本块数量最少，仅为 1 个。

问题二的建模难度显著高于问题一，因为引入了资源共享的因素，即不在同一控制流程中的基本块可以共享 HASH 资源和 ALU 资源。精确刻画考虑了资源共享因素的资源约束极为困难。为了准确刻画该约束，我们首先设计了一套高效地识别同一控

制流程中的基本块组的算法工具包，该工具包包括 **可行路判别算法**、**网络缩减算法** 和 **同流程基本块组提取算法**。以上三种算法通过删除基本块邻接网络中可被替代的边，仅保留最精简的图，从而将 **求解速度提高了 1000 倍以上**。基于此，我们建立了问题二的整数规划模型。由于资源共享条件下的资源约束数量是指数级增长的 (最坏情况下的数量为 2^{607})，因此直接将这些约束全部添加进模型是不实际的。为了解决这个挑战，我们设计了一种精确的 **松弛和约束生成-可行性证明定界算法**。该算法首先将资源约束松弛，快速获得松弛模型的整数解。若该解不是可行解，则调用约束生成算法，使用同流程基本块组提取算法工具包辅助产生资源约束，并将其加回到模型中。此外，并行的模型可以通过目标松弛的方法快速证明可行性或者得到整数可行解，并调用 **可行性证明定界算法** 来快速提升全局上界和下界。该算法在问题二的算例求解中表现出了惊人的效果：算法仅用**183 秒** 就获得了值为**39** 的 **全局最优解 (Gap=0)**。整个求解过程，**约束生成过程被调用 2 次，一共仅生成 11628 个资源约束**。最优分配方案中，平均每一级分配 16 个基本块，第 20 级被分配的基本块最多，为 37 个；第 5 级和第 7 级被分配的基本块数量最少，仅为 1 个。可见，考虑了共享资源的因素后，所使用的的级数大大减少，资源利用率显著提升。

总体来讲，本文针对问题一和问题二的问题特性，设计了一系列的高效加速算法，显著地提高了求解效率，大大降低了求解的复杂度。本文的提出的算法均在合理时间或在短时间内得到了全局最优解，求解效率显著地超过了商业求解器。

关键词：芯片资源排布；整数规划；松弛和约束生成；目标松弛诱导和可行性证明定界；变量固定；网络缩减；广度前向传播

目录

1. **问题重述** 5
1.1 问题背景 5
1.2 问题提出 5
2. **模型假设** 6
3. **符号说明** 6
4. **问题一模型建立与求解** 8
4.1 问题一分析 8
4.2 获取控制依赖：最大可达网络流模型 8
4.2.1 模型建立 10
4.2.2 模型复杂度分析 12
4.2.3 算法设计 12
4.2.4 算法结果与复杂性分析 14
4.3 获取数据依赖 16
4.3.1 算法设计 16
4.3.2 算法结果与复杂性分析 17
4.4 问题一数学模型：芯片资源排布整数规划模型 17
4.5 问题一模型复杂度分析 20
4.6 精确求解算法设计：目标松弛诱导定界和变量固定驱动的分支切割算法 21
4.6.1 目标松弛诱导的定界 21
4.6.2 目标界限诱导的变量固定 24
4.7 问题一求解结果 24
4.7.1 结果与分析 24
4.7.2 算法有效性和时间复杂度分析 31
5. **问题二模型建立与求解** 33
5.1 问题二分析 33
5.2 同一执行流程的基本块对识别：可行路判别算法 34
5.3 网络缩减算法：缩减基本块邻接网络的规模 37
5.3.1 算法瓶颈 37
5.3.2 算法设计 38
5.3.3 算法结果与复杂性分析 40
5.4 问题二数学模型：考虑资源共享的资源排布整数规划模型 40
5.5 精确求解算法设计：松弛和约束生成-可行性证明定界算法 42
5.6 问题二求解结果 43
5.6.1 结果与分析 43

5.6.2 算法有效性和时间复杂度分析 46

6. 模型评价 **47**

6.1 模型的优点 47

6.2 模型的缺点 48

7. 参考文献 **49**

附录 A 问题一的求解代码 **50**

1.1 run_this.py 50

附录 B 问题一的求解代码 **58**

2.1 run_this.py 58

1. 问题重述

1.1 问题背景

PISA (Protocol Independent Switch Architecture) 架构芯片资源排布问题是指在满足流水线各级资源约束和依赖约束的前提下，将各基本块的资源排布到流水线各级中，以提升芯片资源利用效率的优化问题。随着国际形势的日益复杂，各国在芯片等高端科技领域的竞争逐渐加剧，为提升研发效率，可编程的交换芯片诞生。PISA 是目前主流的可编程交换芯片架构之一，不仅具备可编程功能，还和传统的功能固定的交换芯片处理速率相同，应用前景广阔。[1, 2]

PISA 架构主要包括三个组成部分：报文解析 (parser)，多级的报文处理流水线 (Pipeline Pocket Process)，如图1-1右图所示。本课题聚焦于多级的报文处理流水线过程，研究各基本块在流水线各级如何进行排布的问题。由于各基本块会占用一定的资源信息，所以也是芯片的资源排布问题。每个基本块可以被抽象成 P4 程序流程图中的一个节点，图中的有向边表示基本块间的执行逻辑。在 PISA 架构芯片的实际设计中，往往在流水线各级内和各级之间存在一系列复杂的资源约束和依赖约束，使得问题的建模和求解较为困难。同时，芯片的各类资源十分稀缺，因此设计高效的资源排布算法对于芯片的充分利用和编译器的合理设计非常重要。

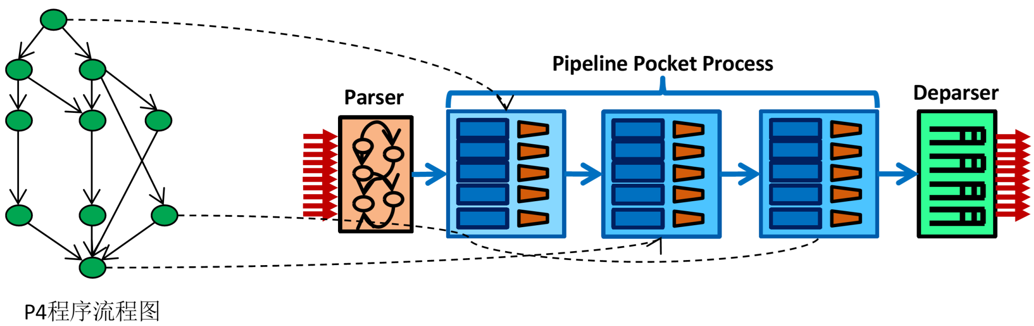


图 1-1 PISA 架构资源排布示意图

1.2 问题提出

本课题旨在满足各基本块的数据依赖、控制依赖、以及各具体子问题的芯片资源约束条件下进行资源排布，建立数学规划模型，设计资源排布算法，以求最大化芯片资源利用率。

问题一： 建立资源排布问题的数学规划模型，以占用的流水线级数尽量短为优化目标，要求满足下述资源约束条件：

- 资源上限约束：流水线每级的 TCAM/HASH/ALU/QUALIFY 资源最大为 1/2/56/64；
- 折叠资源约束：流水线折叠的两级 TCAM 资源加起来最大为 1，HASH 资源加起来最大为 3。(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，从第 32 开始的级数不考虑折叠资源限制)；
- 偶数级数约束：有 TCAM 资源的偶数级数量不超过 5；

- 唯一约束：每个基本块只能排布到一级。

求解建立的数学模型，设计资源排布算法，并按照要求的结果格式输出基本块排布结果。

问题二： 增加执行流程的考虑，规定不在同一条执行流程上的基本块，可以共享 HASH 和 ALU 资源，即如果不同执行流程上的基本块中任意一个的 HASH 资源与 ALU 资源均不超过每级资源限制，这些基本块就可以排布到同一级。基于新增规定，对问题一中的部分资源上限约束和折叠资源约束进行如下更改：

- 资源上限约束：流水线每级中同一条执行流程上的基本块的 HASH/ALU 资源之和最大为 2/56；
- 折叠资源约束：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。

其他约束条件和问题一保持一致，只更改上述约束后重新建立排布优化模型，设计排布算法，按规定格式输出基本块排布结果。

2. 模型假设

本文依据题意进行如下假设：

1. 每个流水级是同质的，即在满足各约束的条件下，基本块可以排布至任意流水级。
2. 基本块之间在满足约束的条件下可以任意组合。
3. 每个基本块只能排布到一个流水级中。
4. 对于邻接矩阵中的任意基本块，当且仅当另一基本块是可达的，二者才能存在依赖关系。

3. 符号说明

本文中涉及的集合、索引、模型参数和决策变量的定义如下所示：

表 1　索引

索引	含义
i, j, b	基本块 $i, j, b \in \mathcal{B}$
r	资源 $r \in \mathcal{R}$
k	层级 $k \in \mathcal{K}$

表 2 集合

集合	含义
$\mathcal{B}$	基本块的集合，$\mathcal{B}=\{0,1,\cdots,B\}$
$\mathcal{R}$	资源的类型集合，$\mathcal{R}=\{TCAM,HASH,ALU,QUALIFY\}$
$\mathcal{K}$	层级的集合，$\mathcal{K}=\{0,1,\cdots,K\}$
$\mathcal{G}=(\mathcal{V},\mathcal{A})$	整个网络图；其中 $\mathcal{V}=\mathcal{B}$ 表示网络图中的节点的集合，$\mathcal{A}$ 表示图中的弧段的集合

表 3 模型参数

参数	含义
B	基本块的数量
a_{ij}	如果基本块 i 和基本块 j 之间存在邻接关系（控制依赖），则 $a_{ij}=1$，否则 $a_{ij}=0$，其中 $i,j\in\mathcal{B}$
b_{ij}	如果基本块 i 和基本块 j 之间存在写后读、写后写依赖关系（关系是 $i<j$），则 $b_{ij}=1$，否则 $b_{ij}=0$，其中 $i,j\in\mathcal{B}$
c_{ij}	如果基本块 i 和基本块 j 之间存在读后写（关系是 $i\leqslant j$）依赖关系，则 $c_{ij}=1$，否则 $c_{ij}=0$，其中 $i,j\in\mathcal{B}$
d_{br}	表示基本块 b 占用资源 $r\in\mathcal{R}$ 的数量
U_r	表示每一层级的资源 $r\in\mathcal{R}$ 的最大使用量
Q_r	m 表示折叠层级的资源 $r\in\{TCAM,HASH\}$ 的最大使用量
O_i	网络中点 i 的出度
T	虚拟节点
F	从虚拟源点流出的总流量，取值为一个足够大的正数，例如 1000000
c	弧段上单位流量的成本

表 4 决策变量

决策变量	含义
y_k	表示第 k 级被使用，$k\in\mathcal{K}$
x_{bk}	0-1 变量，若基本块 $b\in\mathcal{B}$ 被分配到第 k 级，则 $x_{bk}=1$，否则 $x_{bk}=0$
z_k	0-1 变量，表示第 k 级是否占用 TCAM 资源，$k\in\mathcal{K}$ 且 k 为偶数

x_{ij}	非负连续变量；表示弧 (i,j) 上的流量

4. 问题一模型建立与求解

4.1 问题一分析

问题一解决流水线资源排布问题，优化目标为占用的流水线级数尽量短，即最小化最终排布了所有基本块的流水线总级数，同时还要满足 ①控制依赖，②数据依赖以及③资源约束等一系列约束条件。以往研究中 PISA 架构芯片资源排布问题的直接研究较少，而通过对问题的深入研究，此问题可以看作是运筹学中经典的流水作业调度问题 (Flow-shop scheduling problem) 的一个变种。在流水作业调度问题中，一组工件要被分配到不同的机器上，工件的工序间同样存在各类逻辑约束 [3]。类比流水线资源排布问题，一个基本块就可以看作一个工件，然后不同的层级就是不同的机器。流水作业调度问题往往通过构建混合整数规划模型后调用求解器、设计精确算法或者直接采用启发式算法求解 [4]。因此，本文这里将此流水线资源排布问题建模为一个整数规划模型，首先设置一个层级是否被使用的变量，在满足各约束条件的基础上，使得被使用层级总和最小化。

题中资源排布问题的复杂度主要在于流水线各级内和各级之间错综复杂的约束关系，本文分别对下述三类约束关系进行了细致地刻画和考量，搭建模型并设计算法从数据信息中对基本块间的约束关系进行提取，最终构建资源排布模型的约束条件。为了从数据中获取各约束，我们分别进行如下操作：

针对①**控制依赖约束**，基于题目所给邻接基本块数据信息，本文首先构建 P4 程序流程图，并增设虚拟节点搭建**最大可达网络流模型**，设计**广度前向传播算法**进行求解，提取基本块间的控制依赖关系，然后基于控制依赖关系设置第一类基本块间的流水线层级先后约束。

针对②**数据依赖约束**，本文基于各基本块读写的变量信息提取基本块间数据依赖关系，包括读后写、写后读以及写后写三种，然后同样由数据依赖关系生成第二类和第三类基本块间的流水线层级先后约束。

针对③**资源约束**，读取题目所给的各基本块使用的资源信息，构建资源上限约束、折叠资源约束、偶数级数约束以及唯一约束。

基于构建的目标函数和约束条件，本文建立**芯片资源排布整数规划模型**，并设计高效的精确求解算法：**目标松弛诱导定界和变量固定驱动的分支切割算法**。该算法显著加速模型求解，并获得**全局最优解**。

4.2 获取控制依赖：最大可达网络流模型

基于题目给出的邻接基本块数据信息，我们先将每个基本块抽象为一个节点并构建出包含全部基本块（节点）的 P4 流程图（如图4-1a）。我们希望在这张有向无环图上找到每一个具有控制依赖关系的节点对。控制依赖的定义是：当从某个节点出发的路径，只有部分经过下游的另一个节点，两节点构成控制依赖。我们发现，如果从**网络流**

的视角来看，控制依赖在网络流模型中等价于如下判断：

- 如果从源节点 A 流出的流只有一部分流入目标节点 B，则 A 与 B 存在控制依赖。
- 如果从源节点 A 流出的流全部流入目标节点 B，则 A 与 B 不存在控制依赖。
- 如果从源节点 A 流出的流完全不流入目标节点 B，则说明 A 无法达到 B，则 A 与 B 也不存在控制依赖

基于这三条判断，我们将该网络建立为一个等价的**最大可达网络流模型**，并提出了**最大可达网络流线性规划模型**对网络流问题进行刻画，同时设计了**广度前向传播算法**进行求解获得每个节点的流量值，最后通过上述三条判断获得了对每个基本块存在控制关系的基本块集合。

我们首先找到 P4 流程图中所有叶子节点，即4-1b 中的节点 5 和节点 7。增设**虚拟节点 (dummy node)**，添加由叶子节点指向虚拟节点的有向边，生成图4-1b。我们以节点 0 为例来寻找与节点 0 形成控制依赖的节点对，采用最大可达网络流思想进行探索。首先为目标节点 0 设置一个较大的输入流量 100，输入的流量会在网络流图中随有向边向前传播，每个节点的流出流量平分到各边，生成的最大可达网络流图如图4-1c 所示。接下来计算流经的每个节点的流入流量，当流入流量等于目标节点 0 的初始输入流量时，该节点和目标节点不存在控制依赖关系，如图4-1c 中的节点 4。若流入流量不等于初始输入流量，则该节点和目标节点存在控制依赖关系，如图中的节点 1、2、3、5、6、7。

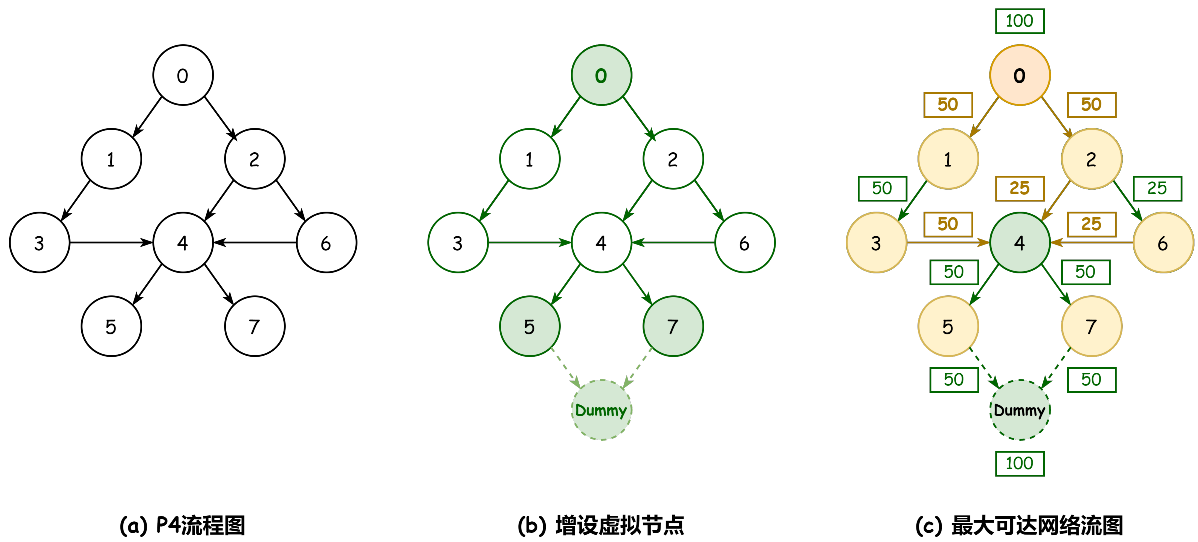


图 4-1　最大可达网络流生成示意图

算法目标：我们希望针对程序控制流网络设计算法，求解最大可达网络流问题，并最终获得每个节点的流量数据。根据控制依赖的定义，我们只需通过比较**源节点**与**目标节点**之间的流量大小关系，就可以得知源节点与目标节点是否存在控制依赖。最后我们遍历所有基本块，每次将一个基本块设置为源节点，再遍历所有其他基本块，每次将一个其他基本块设置为目标节点

算法产出：为每个基本块添加一个属性，该属性为一个列表数据结构，其中存储了与该基本块构成控制依赖的所有其他基本块。

4.2.1 模型建立

对最大可达网络流问题，本节将建立数学规划模型，对模型目标与模型约束进行精确刻画：

模型参数

表 5　模型的输入参数

符号	含义
$\mathcal{B}$	基本块的集合
$\mathcal{G}$	$=(\mathcal{V},\mathcal{A})$，表示整个网络图；其中 $\mathcal{V}=\mathcal{B}$ 表示网络图中的节点的集合，$\mathcal{A}$ 表示图中的弧段的集合
O_i	表示在网络中，点 i 的出度
t	虚拟终点
F	表示从虚拟源点流出的总流量，取值为一个足够大的正数，例如 1000000
c	表示弧段上单位流量的成本

决策变量

- χ_{ij}: 非负连续变量；表示弧 (i,j) 上的流量。

目标函数

目标函数为最大化网络中的流量产生的成本，即

$$\max \quad \sum_{(i,j)\in\mathcal{A}} c\chi_{ij} \tag{1}$$

约束条件

- **约束 1**：首先是从源点 s 出发的流量要平分总流量。

$$\chi_{sj}=\frac{F}{O_s}, \qquad \forall (s,j)\in\mathcal{A}. \tag{2}$$

- **约束 2**：到达虚拟终点的流量总和为 F。

$$\sum_{(j,T)\in\mathcal{A}} \chi_{jT}=F. \tag{3}$$

- **约束 3：流平衡约束**。即对每个中间节点，流入的量等于流出的量。

$$\sum_{(i,j)\in\mathcal{A}} \chi_{ij}-\sum_{(j,i)\in\mathcal{A}} \chi_{ji}=0, \qquad \forall i\in\mathcal{B}. \tag{4}$$

- **约束 4-5：逻辑约束**。如果一个点的流入量大于等于 0（或者大于等于 1），则该点对应的 $w_i = 1$, 否则为 0。

$$\text{if } \sum_{i,j} \chi_{ij} \geqslant 1, \text{ then } w_i = 1, \qquad \forall i \in \mathcal{B}, \tag{5}$$

$$\text{if } \sum_{i,j} \chi_{ij} \leqslant 0, \text{ then } w_i = 0, \qquad \forall i \in \mathcal{B}. \tag{6}$$

- **约束 6：逻辑约束**。一个节点的 $w_i = 1$，则离开该点的弧段的流量必须大于等于 1。

$$\text{if } w_i = 1, \text{ then } \chi_{ij} \geqslant 1, \qquad \forall i \in \mathcal{B}. \tag{7}$$

- **约束 7**：流出节点 i 的流量平分流入节点 i 的总流量。

$$\chi_{ij} = \frac{\sum_{j,i} \chi_{ji}}{O_i}, \qquad \forall i \in \mathcal{B}. \tag{8}$$

- **约束 8**：决策变量的类型约束。

$$\chi_{ij} \geqslant 0, w_i \in \{0, 1\}, \qquad \forall i \in \mathcal{B}, \forall (i,j) \in \mathcal{A}. \tag{9}$$

基于上述构建的目标函数和约束条件，我们得到最大可达网络流混合整数规划模型（Maximum Accessible Network Flow Model, MANFM-MIP）：

$$\begin{aligned}
\max \quad & \sum_{(i,j)\in\mathcal{A}} c\chi_{ij} && && (10)\\
s.t. \quad & \chi_{sj} = \frac{F}{O_s}, && \forall (s,j) \in \mathcal{A}, && (11)\\
& \sum_{(i,j)\in\mathcal{A}} \chi_{ij} - \sum_{(j,i)\in\mathcal{A}} \chi_{ji} = 0, && \forall i \in \mathcal{B}, && (12)\\
& \sum_{(j,T)\in\mathcal{A}} \chi_{jT} = F, && && (13)\\
& \text{if } \sum_{i,j} \chi_{ij} \geqslant 1, \text{ then } w_i = 1, && \forall i \in \mathcal{B}, && (14)\\
& \text{if } \sum_{i,j} \chi_{ij} \leqslant 0, \text{ then } w_i = 0, && \forall i \in \mathcal{B}, && (15)\\
& \text{if } w_i = 1, \text{ then } \chi_{ij} \geqslant 1, && \forall i \in \mathcal{B}, && (16)\\
& \chi_{ij} = \frac{\sum_{j,i} \chi_{ji}}{O_i}, && \forall i \in \mathcal{B}, && (17)\\
& \chi_{ij} \geqslant 0, w_i \in \{0, 1\}, && \forall i \in \mathcal{B}, \forall (i,j) \in \mathcal{A}. && (18)
\end{aligned}$$

模型 (10)-(18) 中含有 0-1 变量 w_i。但是观察到约束 (11) 和约束 (17) 可以自动保证流量大于 0 的限制，因此决策变量 w 以及其相关约束可以直接删去。

我们将模型 (10)-(18) 进行精炼，变成如下的等价线性规划 (Linear Programming) 模型。

$$
\begin{aligned}
\max \quad & 0 && && (19)\\
s.t. \quad & \chi_{sj} = \frac{F}{O_s}, && \forall (s,j) \in \mathcal{A}, && (20)\\
& \sum_{(i,j)\in\mathcal{A}} \chi_{ij} - \sum_{(j,i)\in\mathcal{A}} \chi_{ji} = 0, && \forall i \in \mathcal{B}, && (21)\\
& \sum_{(j,T)\in\mathcal{A}} \chi_{jT} = F, && && (22)\\
& \chi_{ij} = \frac{\sum_{ji} \chi_{ji}}{O_i}, && \forall i \in \mathcal{B}, && (23)\\
& \chi_{ij} \geqslant 0, && \forall i \in \mathcal{B}, \forall (i,j) \in \mathcal{A}. && (24)
\end{aligned}
$$

我们称模型 (19)-(24) 为最大可达网络流线性规划模型（Maximum Accessible Network Flow Model, MANFM-LP）。

4.2.2 模型复杂度分析

我们分别对最大可达网络流模型在混合整数规划与线性规划两种版本的模型复杂度进行分别分析。这两种模型在 **0-1 变量数量**，**连续变量数量**以及**约束数量**上的差异如表6所示。其中混合整数规模型的 0-1 变量数量与约束数量均多于线性规划模型。

表 6　最大可达网络流的两种模型复杂度分析

模型名称	IMANFM-MIP	LMANFM-LP
模型内容	Eq. (1)-(9)	Eq. (19)-(24)
模型类型	混合整数规划	线性规划
0-1 变量数量	$\|\mathcal{B}\|$	0
连续变量数量	$\|\mathcal{A}\|$	$\|\mathcal{A}\|$
约束数量	$6\|\mathcal{B}\| + \|\mathcal{A}\|$	$2\|\mathcal{B}\| + \|\mathcal{A}\|$

4.2.3 算法设计

（1）数据预处理

为了求解 4.2.1 中提出的最大可达网络流模型，本文提出了相应的算法进行实现。但是实现该算法依赖于程序控制流的有向无环网络图，因此需要先进行相应的数据预

处理。具体操作是先建立包含所有基本块的网络节点，并从 attachment3.csv 文件中读取所有基本块的邻接关系，根据邻接关系依次添加有向边，从而完成有向无环图的构建。

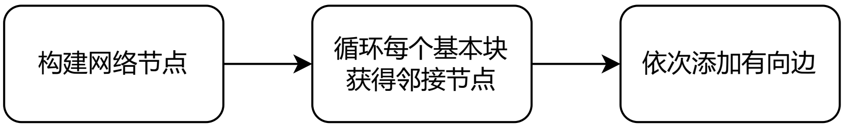


图 4-2　控制依赖算法预处理流程

（2）算法流程

基于预处理得到的有向无环图，本文提出了广度前向传播算法。我们基于广度优先思想遍历所有节点，并通过父节点的流入量分配子节点的流出量，实现了流量从源节点向所有可达节点的前向传播。最后依据节点流量情况，依次检查源节点与目标节点之间的控制依赖关系，如果满足，则对源节点基本块的属性进行更新。据此，算法的具体流程如下：

Algorithm 1 广度前向传播算法

Require: 网络图 $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, 起点 s，终点 t

Ensure: 基本块之间的依赖关系字典 $\mathcal{D}$

总流量 $F \leftarrow$ 足够大的正数，初始化起始点的总流量为 $s.flow \leftarrow F$，$k \leftarrow 1$, 初始化起始点的序号 $s.ID \leftarrow k$，初始化未探索的子节点集合 $\mathcal{Q} \leftarrow \{s\}$，初始化基本块之间的依赖关系字典 $\mathcal{D} \leftarrow \varnothing$

/* **广度前向传播：设置流量** */

while $\mathcal{Q}$ 非空 **do**

　　当前节点 $v \leftarrow \arg\min_{v}\{v.ID|v \in \mathcal{Q}\}$ (广度优先)

　　更新 $\mathcal{Q}$，$\mathcal{Q} \leftarrow \mathcal{Q} \backslash \{v\}$

　　计算进入点 v 的流量 $v.flow$

　　for 每个 v 的子节点 u **do**

　　　　设置边 $e = (v, u)$ 的流量 $f_e \leftarrow \frac{v.flow}{O_v}$

　　　　更新 k，$k \leftarrow k + 1$

　　　　设置点的序号 $u.ID \leftarrow k$

　　　　更新 $\mathcal{Q}$，$\mathcal{Q} \leftarrow \mathcal{Q} \cup \{u\}$

　　end for

end while

/* **检查依赖关系** */

for 每个点 v **do**

　　for 每个点 u **do**

　　　　if $v.flow \neq u.flow$ **then**

　　　　　　点 v 和 u 之间有依赖，更新 $\mathcal{D} \leftarrow \mathcal{D} \cup \{(v, u) : 1\}$

　　　　end if

　　end for

end for
return $\mathcal{D}$

4.2.4 算法结果与复杂性分析

（1）算法结果展示

我们基于上述算法，对 4.2.1 节中的模型进行编程并求解，并获得所有基本块的控制依赖关系。由于本问题涉及 607 个基本块，数据量较大不便于一次性进行展示，本节中我们只提取一部分数据进行展示，全部详细结果请见附件 controlDependence.csv。

在表7中，第 1 列是基本块的编号；第 2 列至倒数第 2 列是与该基本块存在控制依赖的所有其他基本块编号；最后一列是这些基本块的总数量。由于篇幅限制，我们仅在表7中展示了前 20 个与最后 20 个基本块的控制依赖求解结果。同时，如果对每个基本块存在控制依赖的其他基本块少于等于 10 个，则这些基本块会全显示；若多于 10 个，我们只展示编号由小到大排列的前 5 个与最后 5 个基本块。

表 7　各基本块控制依赖关系的求解结果（部分）

Block#	与该基本块存在控制依赖的其他基本块（部分）										总数
Block0	1	-	-	-	-	-	-	-	-	-	1
Block1	-	-	-	-	-	-	-	-	-	-	0
Block2	-	-	-	-	-	-	-	-	-	-	0
Block3	1	586	587	588	589	590	591	-	-	-	7
Block4	1	586	587	588	589	590	591	-	-	-	7
Block5	0	1	2	6	7	$\cdots$	351	352	353	354	355	74
Block6	0	1	2	7	23	$\cdots$	351	352	353	354	355	73
Block7	0	1	2	23	24	$\cdots$	351	352	353	354	355	72
Block8	0	1	2	23	24	$\cdots$	351	352	353	354	355	72
Block9	0	1	2	23	24	$\cdots$	351	352	353	354	355	72
Block10	0	1	2	23	24	$\cdots$	351	352	353	354	355	72
Block11	0	1	2	3	4	$\cdots$	602	603	604	605	606	548
Block12	0	1	2	3	4	$\cdots$	602	603	604	605	606	548
Block13	0	1	2	3	4	$\cdots$	602	603	604	605	606	568
Block14	0	1	2	3	4	$\cdots$	602	603	604	605	606	568
Block15	0	1	2	3	4	$\cdots$	602	603	604	605	606	569
Block16	0	1	2	3	4	$\cdots$	602	603	604	605	606	568

Block#	与该基本块存在控制依赖的其他基本块（部分）											总数
Block17	0	1	2	3	4	···	602	603	604	605	606	568
Block18	0	1	2	3	4	···	602	603	604	605	606	568
Block19	0	1	2	3	4	···	602	603	604	605	606	568
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Block587	1	588	-	-	-	-	-	-	-	-	-	2
Block588	1	-	-	-	-	-	-	-	-	-	-	1
Block589	1	588	-	-	-	-	-	-	-	-	-	2
Block590	1	588	-	-	-	-	-	-	-	-	-	2
Block591	1	588	-	-	-	-	-	-	-	-	-	2
Block592	0	1	2	5	6	···	432	593	604	605	606	133
Block593	0	1	2	5	6	···	431	432	604	605	606	132
Block594	0	1	2	5	6	···	431	432	604	605	606	132
Block595	0	1	2	5	6	···	599	600	604	605	606	383
Block596	0	1	2	5	6	···	599	600	604	605	606	382
Block597	0	1	2	5	6	···	599	600	604	605	606	381
Block598	0	1	2	5	6	···	599	600	604	605	606	380
Block599	0	1	2	5	6	···	594	600	604	605	606	379
Block600	0	1	2	5	6	···	593	594	604	605	606	378
Block601	0	1	2	5	6	···	593	594	604	605	606	378
Block602	0	1	2	5	6	···	593	594	604	605	606	378
Block603	0	1	2	5	6	···	593	594	604	605	606	378
Block604	0	1	2	5	6	···	352	353	354	355	605	86
Block605	0	1	2	5	6	···	351	352	353	354	355	85
Block606	0	1	2	5	6	···	351	352	353	354	355	85

（2）算法复杂度分析

广度前向传播算法的实现思路较为直观，但在具体编程实现上具有许多难点，具体体现在 1、需要设计特定数据结构用于保存有向无环图的节点与边信息；2、需要设计算法快速识别每个节点的父节点与子节点；3，需要实现 Ford-Fulkerson 算法求解最大流问题。

该算法对节点的遍历取决于有向图的结构，有向图结构的复杂性会直接影响算法复杂性。**在最坏情况下，模型要先遍历所有节点，对于每个节点通过广度优先再次遍历所有其他节点，因此最坏情况算法复杂度为 $\mathcal{O}(n^2)$。而在最好情况下，模型的复杂度为 $\mathcal{O}(n)$。算法在 23.7 秒成功了所有基本块的控制依赖关系**。

4.3 获取数据依赖

与控制依赖类似的，我们希望通过读取模型文件 attachment2.csv 中每个基本块对不同变量的读写情况，获得对于每个基本块存读后写、写后读、写后写三种数据依赖关系。最终照控制依赖相同的方式，在编程实现上对每个基本块分别定义三种数据依赖的属性，每个属性中保存了与之存在依赖关系的所有其他基本块。

4.3.1 算法设计

获取读后写、写后读、写后写三种数据依赖的算法流程如图4-3所示。本问题中，获取数据依赖的难点在于可达性判断，即对于每一个基本块，我们需要遍历所有其他基本块，先判断是否满足可达性条件，再判断是否属于读后写、写后读、写后写三种数据依赖。如果均为是，则我们将会对基本块的属性进行更新，记录与每个基本块构成不同数据依赖的其他基本块。

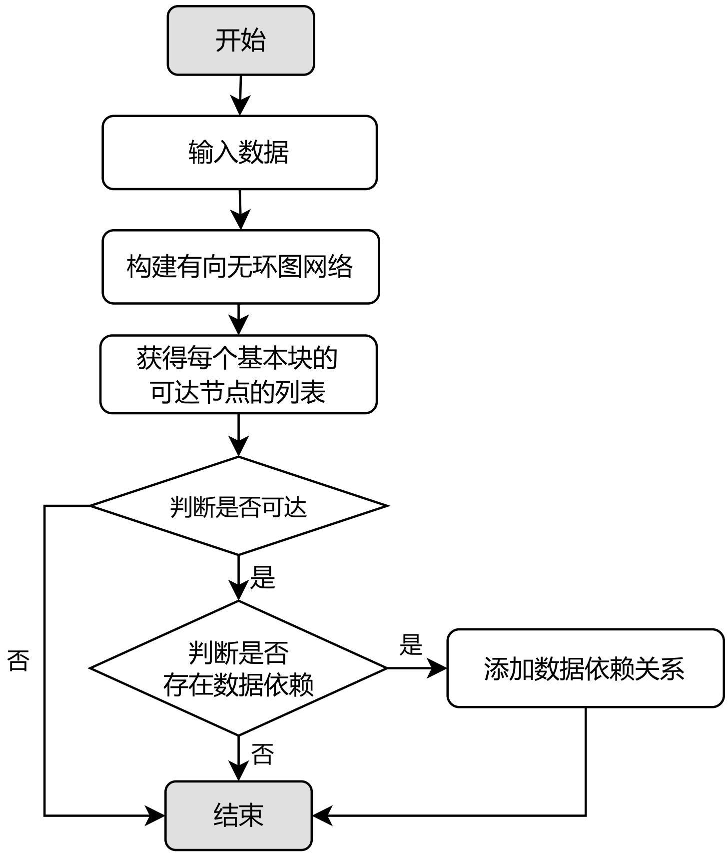


图 4-3　数据依赖算法流程

4.3.2 算法结果与复杂性分析

由于篇幅的限制，我们这里不对三种数据依赖关系进行完整的展示。关于先读后写依赖、先写后读依赖、先写后写依赖的完整算法求解结果请分别见附件 readWriteDependence.csv，writeReadDependence.csv，writeWriteDependence.csv 中的内容。由于我们使用了 dijstra 算法，通过识别是否存在最短路的方式判断是否可达，同时我们需要对所有基本块对进行识别，对所有起点与终点进行遍历，因此算法复杂度为 $\mathcal{O}(N^2(VlogN))$，其中 V 是边的数量，N 是节点数量。

4.4 问题一数学模型：芯片资源排布整数规划模型

(1) 输入参数

表 8　模型的输入参数

符号	含义
B	基本块的数量
$\mathcal{B}$	基本块的集合，$\mathcal{B} = \{0, 1, \cdots, B\}$
a_{ij}	如果基本块 i 和基本块 j 之间存在邻接关系（控制依赖），则 $a_{ij} = 1$，否则 $a_{ij} = 0$，其中 $i, j \in \mathcal{B}$
b_{ij}	如果基本块 i 和基本块 j 之间存在写后读、写后写依赖关系（关系是 $i < j$），则 $b_{ij} = 1$，否则 $b_{ij} = 0$，其中 $i, j \in \mathcal{B}$
c_{ij}	如果基本块 i 和基本块 j 之间存在读后写（关系是 $i \leqslant j$）依赖关系，则 $c_{ij} = 1$，否则 $c_{ij} = 0$，其中 $i, j \in \mathcal{B}$
$\mathcal{R}$	资源的类型集合，本题目中 $\mathcal{R} = \{TCAM, HASH, ALU, QUALIFY\}$
	表示基本块 b 占用资源 $r \in \mathcal{R}$ 的数量
U_r	表示每一层级的资源 $r \in \mathcal{R}$ 的最大使用量
Q_r	表示折叠层级的资源 $r \in \{TCAM, HASH\}$ 的最大使用量

(2) 决策变量

- y_k: 表示第 k 级被使用，$k \in \mathcal{K}$；
- x_{bk}: 0-1 变量，若基本块 $b \in \mathcal{B}$ 被分配到第 k 级，则 $x_{bk} = 1$，否则 $x_{bk} = 0$；
- z_k: 0-1 变量，表示第 k 级是否占用 TCAM 资源，$k \in \mathcal{K}$ 且 k 为偶数。

(3) 目标函数

本问题的目标函数为最小化所使用的的层级。

$$\min \sum_{k \in \mathcal{K}} y_k \tag{25}$$

（4）**约束条件**

约束 (1)：首先，只有第 k 级被使用了，第 $k+1$ 级才允许被使用。

$$y_k \geqslant y_{k+1}, \qquad \forall k \in \mathcal{K} \backslash \{K\}. \tag{26}$$

约束 (2)：唯一约束。每一个基本块必须被分配且只分配到一个层级中。

$$\sum_{k \in \mathcal{K}} x_{bk} = 1, \qquad \forall b \in \mathcal{B}. \tag{27}$$

约束 (3)-(4)：如果一个级没有被启用，则不会有任何基本块被分配（公式 (41)）。如果一个级被启用了，则一定至少有一个基本块被分到该级（公式 (42)）。其中 M 是一个足够大的正数，可以通过分析设置一个最合适的值。

$$\sum_{b \in \mathcal{B}} x_{bk} \leqslant M y_k, \qquad \forall k \in \mathcal{K}, \tag{28}$$

$$1 - \sum_{b \in \mathcal{B}} x_{bk} - M(1 - y_k) \leqslant 0, \qquad \forall k \in \mathcal{K}. \tag{29}$$

约束 (5)：资源上限约束。每一级占用的资源数量要小于等于最大的允许值。

$$\sum_{b \in \mathcal{B}} d_{br} x_{bk} \leqslant U_r, \qquad \forall k \in \mathcal{K}, \forall r \in \mathcal{R}. \tag{30}$$

约束 (6)：折叠资源约束。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，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。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。

$$\sum_{b \in \mathcal{B}} (d_{br} x_{bk} + d_{br} x_{b,k+16}) \leqslant Q_r, \quad \forall k \in \{0, 1, \cdots, 15\}, \forall r \in \{TCAM, HASH\}. \tag{31}$$

其中 $Q_{TCAM} = 1, Q_{HASH} = 3.$

约束 (7)-(8)：偶数级数约束。TCAM 的资源占用约束，即占用 TCAM 资源的偶数级数量不超过 5。首先需要引入 0-1 变量表示一个级 k 是否占用了 TCAM 资源，然后再限制最多有 5 级占用 TCAM。这些限制可以用公式 (45) 和 (46) 实现。公式 (45) 表示如果偶数级占用了 TACM 资源，则 $z_k = 1$。公式 (46) 表示如果偶数级占用了占用 TCAM 的偶数级的数量不大于 5。

$$\text{if } \sum_{b \in \mathcal{B}} d_{b,\,TCAM}\ x_{bk} \geqslant 1, \text{ then } z_k = 1, \qquad \forall k \text{ 是偶数}, \tag{32}$$

$$\sum_{k\in\mathcal{K}} z_k \leqslant 5. \tag{33}$$

接下来是数据依赖关系和控制依赖关系相关的逻辑约束。

约束 (9): 控制依赖关系。若基本块 i 和 j 存在控制依赖关系，则二者的级数一定满足 i的级数 $\leqslant j$的级数。这里需要借助广度前向传播算法得到的依赖关系结果（算法1 ）。如果二者存在控制依赖，即 $a_{ij}=1$，则二者的级数一定满足 i的级数 $\leqslant j$的级数。该约束可以表达为公式 (47)。

$$\sum_{k\in\mathcal{K}} k\cdot x_{ik} \leqslant \sum_{k\in\mathcal{K}} k\cdot x_{jk}, \qquad \forall i,j\in\mathcal{B}, \text{and} \ \ a_{ij}=1. \tag{34}$$

约束 (10): 写后读和写后写的关系。基本块 i 和 j 有写后读和写后写关系，且从 i 到 j 是可达的 (注意，可达的条件是必须的)，即 $b_{ij}=1$，则一定满足 i的级数 $< j$的级数。我们可以将其转化为带等号的不等式，即 i的级数 $\leqslant j$的级数 -1。该约束可以表达为公式 (48)。

$$\sum_{k\in\mathcal{K}} k\cdot x_{ik} \leqslant \sum_{k\in\mathcal{K}} k\cdot x_{jk} - 1, \qquad \forall i,j\in\mathcal{B} \text{ and } \ b_{ij}=1. \tag{35}$$

约束 (11): 读后写的关系。如果基本块 i 和 j 有读后写关系，且从 i 到 j 是可达的 (注意，可达的条件是必须的)，即 $c_{ij}=1$，则一定满足 i的级数 $\leqslant j$的级数。

$$\sum_{k\in\mathcal{K}} k\cdot x_{ik} \leqslant \sum_{k\in\mathcal{K}} k\cdot x_{jk}, \qquad \forall i,j\in\mathcal{B} \text{ and } \ c_{ij}=1. \tag{36}$$

约束 (12): 变量的类型约束。

$$x_{bk}, y_k, z_k \in \{0,1\}, \qquad \forall b\in\mathcal{B}, \forall k\in\mathcal{K}. \tag{37}$$

基于上述构建的目标函数和约束条件，我们得到模型 (38)-(50) 为芯片资源排布整数规划模型（Chip Resource Allocation Optimization Model, CRAOM-IP）：

$$\min \ \sum_{k\in\mathcal{K}} y_k \tag{38}$$

$$s.t. \quad y_k \geqslant y_{k+1}, \qquad \forall k\in\mathcal{K}\backslash\{K\}. \tag{39}$$

$$\sum_{k\in\mathcal{K}} x_{bk} = 1, \qquad \forall b\in\mathcal{B}. \tag{40}$$

$$\sum_{b\in\mathcal{B}} x_{bk} \leqslant M y_k, \qquad \forall k\in\mathcal{K}, \tag{41}$$

$$1-\sum_{b\in\mathcal{B}} x_{bk} - M(1-y_k) \leqslant 0, \qquad \forall k\in\mathcal{K}. \tag{42}$$

$$\sum_{b\in\mathcal{B}} d_{br}x_{bk} \leqslant U_r, \qquad \forall k\in\mathcal{K}, \forall r\in\mathcal{R}. \tag{43}$$

$$\sum_{b\in\mathcal{B}} (d_{br}x_{bk} + d_{br}x_{b,k+16}) \leqslant Q_r, \qquad \forall k\in\{0,1,\cdots,15\}, \forall r\in\{TCAM, HASH\}. \tag{44}$$

$$\text{if} \sum_{b\in\mathcal{B}} d_{b,\,TCAM}\ x_{bk} \geqslant 1,\ \text{then}\ z_k = 1, \quad \forall k\ \text{是偶数}. \tag{45}$$

$$\sum_{k\in\mathcal{K}} z_k \leqslant 5. \tag{46}$$

$$\sum_{k\in\mathcal{K}} k\cdot x_{ik} \leqslant \sum_{k\in\mathcal{K}} k\cdot x_{jk}, \qquad \forall i,j\in\mathcal{B}, \text{and}\ \ a_{ij}=1. \tag{47}$$

$$\sum_{k\in\mathcal{K}} k\cdot x_{ik} \leqslant \sum_{k\in\mathcal{K}} k\cdot x_{jk} - 1, \qquad \forall i,j\in\mathcal{B}\ \text{and}\ \ b_{ij}=1. \tag{48}$$

$$\sum_{k\in\mathcal{K}} k\cdot x_{ik} \leqslant \sum_{k\in\mathcal{K}} k\cdot x_{jk}, \qquad \forall i,j\in\mathcal{B}\ \text{and}\ \ c_{ij}=1. \tag{49}$$

$$x_{bk}, y_k, z_k \in \{0,1\}, \qquad \forall b\in\mathcal{B}, \forall k\in\mathcal{K}. \tag{50}$$

4.5 问题一模型复杂度分析

模型 (38)-(50) 是一个整数规划模型。且是一个 NP-hard 问题，其模型规模如表所示。

表 9 芯片资源排布整数规划模型（CRAOM-IP）的复杂度

模型名称	**CRAOM-IP**	数量
模型内容	Eq. (38)-(50)	–
模型类型	纯 0-1 整数规划	–
0-1 变量数量	$\|\mathcal{B}\|\|\mathcal{K}\| + 2\|\mathcal{K}\|$	$609\|\mathcal{K}\|$
约束数量	$3\|\mathcal{B}\|^2 + 30 + 7\|\mathcal{K}\| + \|\mathcal{B}\|$	$369086 + 7\|\mathcal{K}\|$
求解难度	NP-Hard	–

针对问题给出的算例，$|\mathcal{B}|$=607，但是 $|\mathcal{K}|$ 是最大可能的级数。但是不幸的是，$|\mathcal{K}|$ 是未知的。为了保证问题可行，我们可以为 $|\mathcal{K}|$ 设置一个足够大的数值，例如设置 $|\mathcal{K}| = |\mathcal{B}| = 607$ 一定是一个可行的方案。但是这种设置过于悲观，会导致问题的规模非常大，难以在合理时间内求解。

为此，本文通过一些算法来设置一个较小的 $|\mathcal{K}|$，从而精炼模型，最大限度地减少决策变量的数量。

4.6 精确求解算法设计：目标松弛诱导定界和变量固定驱动的分支切割算法

本节提出了一种求解 CRAOM-IP 模型的高效精确算法，称之为 **目标松弛诱导定界和变量固定驱动的分支切割算法**。该算法可以显著加速 CRAOM-IP 模型的求解，并且可以获得最优解。

由于本题目的目标函数为 $\sum_{k\in\mathcal{K}} y_k$，其取值非常有限。如果能够获得该目标值的一个比较紧的上界和下界，就可以很容易地进行加速。比如，对变量 y_k 进行固定，对目标函数的界限进行收紧等。

上述算法主要包含 3 个大部分：**目标松弛诱导的定界**、**目标界限诱导的变量固定**、**分支切割算法**。其中，目标松弛诱导定界过程可以有效收紧全局上下界，加速剪枝；目标界限诱导的变量固定可以将特定的决策变量进行值固定，从而缩减规模，得到子整数规划 (Sub-IP)；分支切割算法为求解器的求解部分，本文调用 Gurobi 求解器来完成该部分。图4-4给出了整个算法的框架图。

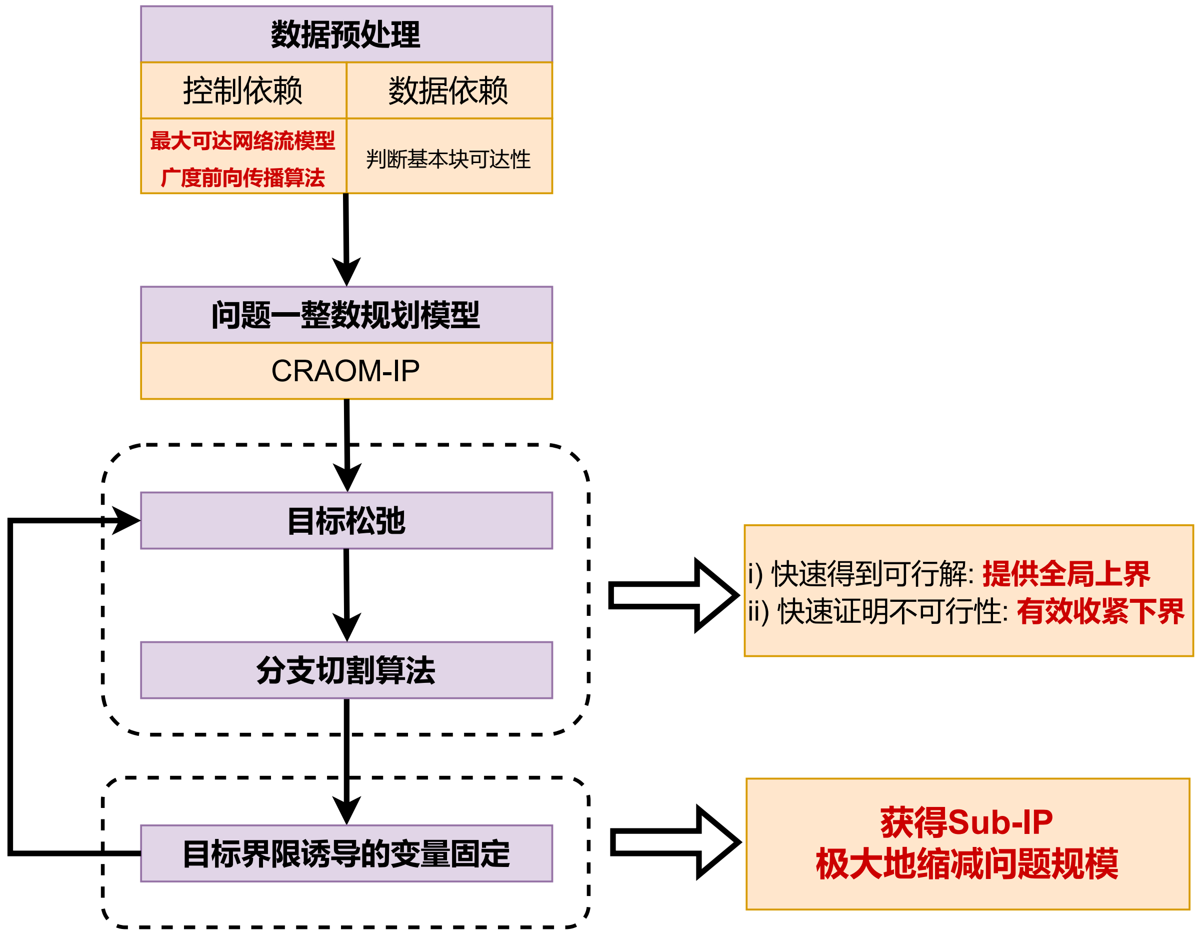


图 4-4 问题一求解算法框架：目标松弛诱导定界和变量固定驱动的分支切割算法

4.6.1 目标松弛诱导的定界

带有目标函数的整数规划模型往往难以求解，但是若将目标函数设置为常数，则整数规划模型变化为获得一个可行解。这种做法通常被称之为零目标 (Zero Objective)。本

文将其称之为 **目标松弛**。执行了目标松弛后的模型变化为

$$
\begin{aligned}
\min \quad & 0 && (51)\\
s.t. \quad & \text{约束}(38)-(50). && (52)\\
& y_k = 0, x_{b,k} = 0, z_k = 0 \qquad \forall k = \{\underline{Z}, \underline{Z}+1, \cdots, |\mathcal{K}|\}, \forall b \in \mathcal{B}. && (53)
\end{aligned}
$$

其中，$\underline{Z}$ 是一个目标函数的推断的下界。**松弛了目标函数的模型 (51)-(52) 求解难度大大降低，因为只需要找到一个可行解即可，不需要找到最优解**。通过设置不同的 $\underline{Z}$，可以达到高效收紧全局上下界的目的。具体包括下面的 2 种情况。

情况 1：获得很紧的下界。若目标松弛模型 (51)-(53) 无可行解，则说明全局的下界 $Z_l \geqslant \underline{Z}+1$。此时可以将 $\underline{Z}$ 增加 1，再次求解松弛模型 (51)-(53)。

例如，若设置 $\underline{Z}=66$，求解松弛模型 (51)-(53)，得到模型不可行，则说明在 66 级之内，不可能将 607 个基本块排布完成。这说明至少需要 67 个级才能完成。因此，可以得到一个全局的比较紧的下界为 $Z_l = \underline{Z}+1 = 66+1 = 67$。

情况 2：获得很紧的上界。若目标松弛模型 (51)-(53) 获得了可行解，其目标值为 Z'，则说明全局最紧的上界 $Z_u \leqslant Z'$。

例如，若设置 $\underline{Z}=69$，求解松弛模型 (51)-(53)，得到了模型的一个目标值为 68 的解，则说明全局最紧的上界 $Z_u \leqslant 68$，即全局最优解 $Z^* \leqslant 68$。

综上，可得 CRAOM-IP 模型的最优解 Z^* 的取值范围为

$$\underline{Z}+1 = Z_l \leqslant \bar{Z}^* = \sum_{k\in\mathcal{K}} y_k^* \leqslant Z' \leqslant Z^u. \tag{54}$$

通过上述操作，Z^l 和 Z^u 可以作为全局的上界和下界提供给求解 CRAOM-IP 的分支切割算法，以加速剪枝 (prune)，显著减小探索的分支切割树 (Branch and cut tree) 的规模。

Algorithm 2 目标松弛诱导的定界算法

Require: 全局下界的初始推断值 $\underline{Z}$，步长 λ
Ensure: 当前全局较紧的上界 Z_u 和下界 Z_l

初始化 $Z_l \leftarrow$ 求解 CRAOM-IP 的线性松弛获得的下界的向上圆整
初始化 $Z_u \leftarrow |\mathcal{K}|$
构建目标松弛诱导模型 (51)-(52)
while 间隙 Gap>0 **and** 求解时间 $<$ 时间限制 **do**
    求解目标松弛诱导模型 (51)-(52)
    if 模型不可行 **then**
        更新全局下界 $Z_l \leftarrow \underline{Z}+1$
        $\underline{Z} \leftarrow \max\{\underline{Z}-\lambda, Z_l+1\}$
    else if 模型可行 **then**
        更新全局上界 $Z_u \leftarrow \min\{\underline{Z}, Z_u\}$

$\underline{Z} \leftarrow \max\{\underline{Z} - \lambda, Z_l + 1\}$

end if

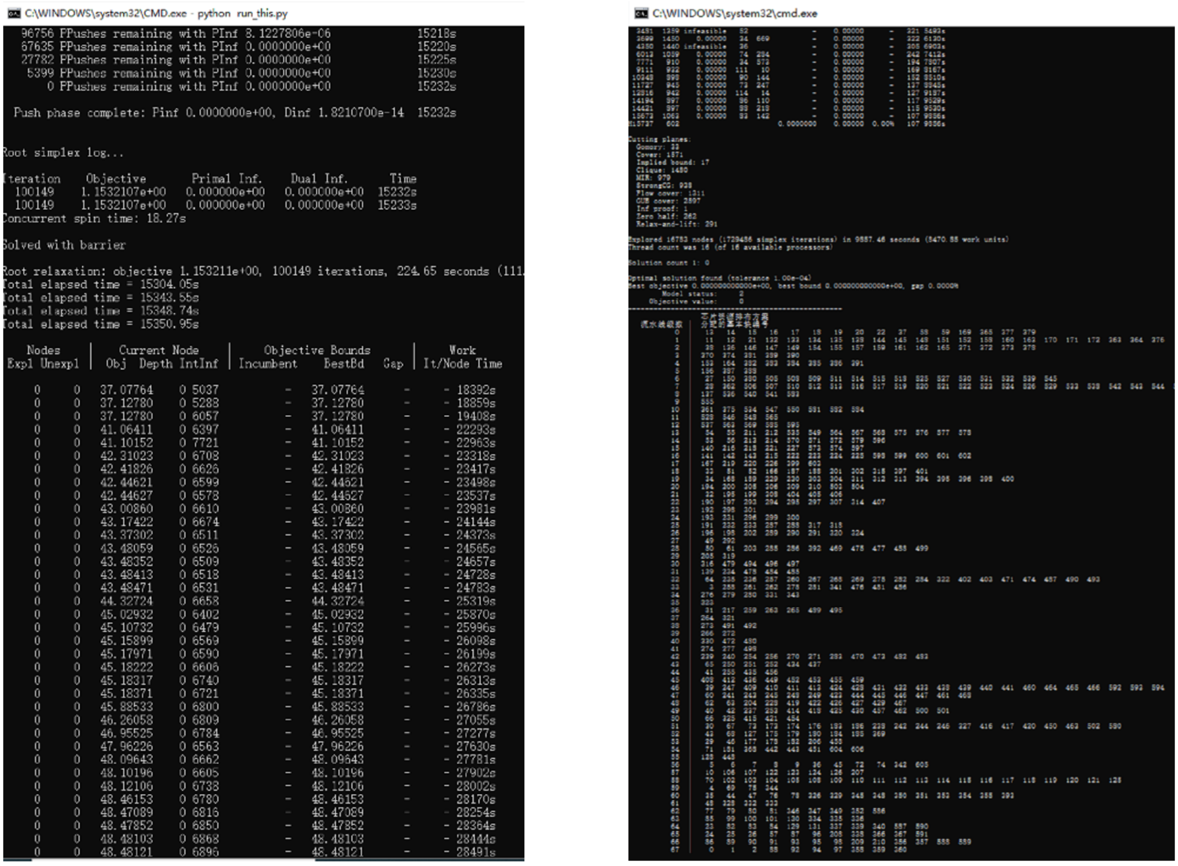
更新步长 λ

end while

return 当前最好的全局上界 Z_u 和下界 Z_l

在题目所给的数据集上的数值实验表明，**目标松弛诱导的定界方法** 在 **5700 秒**以内找到了的下界序列为$Z^l = 61, 62, 64, 66, 67, 68\cdots$，找到的上界序列为$Z^u = 71, 70, 69, 68, \cdots$。可见，目标松弛诱导得到的界限非常紧，解的间隙 (Gap) 的序列相应地为 14%、11%、7.2%、2.9%、1.7%、0%，最终，算法在**1.5 小时**左右得到**目标函数为 68 的最优解**。如图4-5b 所示。

但是**直接调用 Gurobi 求解** CRAOM-IP 模型，在 **8 小时**之内，仍然只能得到**下界 49**，而**没有找到任何可行解**，如图4-5a 所示。(分支切割得到的下界为 48.48121，但由于真实目标函数是整数，因此可以对该数值进行向上圆整，所以目标值的下界为 $\lceil 48.48121 \rceil = 49$)。



(a) 直接调用 Gurobi 求解　　(b) 目标界限诱导的变量固定算法求解

图 4-5　求解结果展示（直接调用求解器 v.s. 精确算法加速）

4.6.2 目标界限诱导的变量固定

基于上述的目标函数的全局上界 Z^u 和下界 Z^l，可以对变量 y_k，x_{bk}，z_k 进行值的固定。这样的变量值固定操作可以极大地减少变量的数量，从而显著地加快问题的求解。即固定如下的变量取值

$$y_k = 1, \qquad \forall k = \{1, 2, \cdots, Z^l\}, \tag{55}$$

$$y_k = 0, \qquad \forall k = \{Z^u, Z^u + 1, \cdots, |\mathcal{K}|\}, \tag{56}$$

$$z_k = 0, \qquad \forall k = \{Z^u, Z^u + 1, \cdots, |\mathcal{K}|\}, \tag{57}$$

$$x_{bk} = 0, \qquad \forall k = \{Z^u, Z^u + 1, \cdots, |\mathcal{K}|\}, \forall b \in \mathcal{B}. \tag{58}$$

根据本题目提供的算例数据，本文通过 目标松弛诱导的定界 方法可以获得很紧的上界和下界。例如，我们可以在 2700 秒以内使用目标松弛诱导的定界算法获得了一个目标函数的上界 $Z_u = 71$。此外，我们可以使用 Gurobi 的分支切割算法获得的最好下界为 $Z_l = 66$，因此，可以根据上述的方法对变量进行固定。通过上述固定方法，可以固定的变量的个数如下表所示。

表 10 目标界限诱导的变量固定的效果 ($Z_l = 66, Z_u = 71, |\mathcal{K}| = 607$)

决策变量	变量总数 (固定前)	固定的值	固定的数量	剩余的量 (固定后)
y_k	607	$y_0 \sim y_{65} = 1, y_{71} \sim y_{\|\mathcal{K}\|} = 0$	601	6
x_{bk}	368449	$x_{bb}, x_{b,71} \sim x_{b,\|\mathcal{K}\|} = 0, \forall b \in \mathcal{B}$	325317	43132
z_k	607	$z_{71} \sim z_{\|\mathcal{K}\|} = 0, z_k = 0, \forall k$是奇数	572	35
总计	369663	–	362490	43173

其他部分，均调用 Gurobi 的分支切割算法求解。

4.7 问题一求解结果

4.7.1 结果与分析

针对问题一，本文设计目标松弛诱导定界和变量固定驱动的分支切割算法，求解芯片资源排布整数规划模型，其中的分支切割算法通过 Python 调用 Gurobi 求解器完成，使用环境为 Python 3.8.11，Gurobi 9.5.1。求解结果如表11所示，表中展示了问题一中各流水级最优的基本块排布方案和各流水级所排布的基本块数量。总流水级数最优解为68。平均每一级分配 9 个基本块，第 7 级被分配的基本块最多，为 34 个；第 9 级和第 35 被分配的基本块数量最少，仅为 1 个。本文对所获得的最优解进行了可行性验证 (Feasibility Checker)，所得结果完全满足资源约束和各类依赖约束，验证结果如图4-6所示。问题一求解结果详见附件问题一结果.xlsx。

表 11　各流水级基本块排布求解结果

总流水级数最优解：68

Level#	Sum	问题一各级最优的基本块排布																
L0	**16**	13	14	15	16	17	18	19	20	22	37	58	59	169	365	377	379	
L1	**22**	11	12	21	132	133	134	135	138	144	145	148	151	152	158	160	163	170
		171	172	363	364	376												
L2	**16**	38	136	146	147	149	154	155	157	159	161	162	165	371	372	373	378	
L3	**5**	370	374	381	389	390												
L4	**8**	153	164	382	383	384	385	386	391									
L5	**3**	156	387	388														
L6	**17**	27	150	380	505	508	509	511	514	515	518	525	527	530	531	532	539	545
L7	**34**	28	362	506	507	510	512	513	516	517	519	520	521	522	523	524	526	529
		533	538	542	543	544	551	552	553	554	556	557	558	559	560	561	562	566
L8	**5**	137	536	540	541	583												
L9	**1**	555																
L10	**8**	361	375	534	547	550	581	582	584									
L11	**4**	528	546	548	565													
L12	**5**	537	563	569	585	595												
L13	**13**	54	55	211	212	535	549	564	567	568	575	576	577	578				
L14	**9**	53	56	213	214	570	571	572	579	596								
L15	**8**	140	216	218	221	227	573	574	597									
L16	**13**	141	142	143	215	222	223	224	225	598	599	600	601	602				
L17	**6**	167	219	220	226	399	603											
L18	**11**	33	51	52	166	187	188	201	302	315	397	401						
L19	**15**	34	168	189	229	230	303	304	311	312	313	394	395	396	398	400		
L20	**8**	194	200	305	306	309	310	503	504									
L21	**7**	32	195	199	308	404	405	406										
L22	**9**	190	197	293	294	295	297	307	314	407								
L23	**3**	192	298	301														

Level#	Sum	问题一各级最优的基本块排布
L24	**5**	193 231 296 299 300
L25	**7**	191 232 233 287 288 317 318
L26	**8**	196 198 202 289 290 291 320 324
L27	**2**	49 292
L28	**11**	50 61 203 285 286 392 469 475 477 488 499
L29	**2**	205 319
L30	**5**	316 479 494 496 497
L31	**5**	139 234 478 484 485
L32	**19**	64 235 236 257 260 267 268 269 275 282 284 322 402 403 471 474 487 490 493
L33	**10**	3 258 261 262 278 281 341 476 481 486
L34	**5**	276 279 280 331 343
L35	**1**	323
L36	**7**	31 217 259 263 265 489 495
L37	**2**	264 321
L38	**3**	273 491 492
L39	**2**	266 272
L40	**3**	330 472 480
L41	**3**	274 277 498
L42	**11**	239 240 254 256 270 271 283 470 473 482 483
L43	**6**	65 250 251 252 434 437
L44	**4**	41 255 435 456
L45	**8**	408 412 436 449 452 453 455 459
L46	**22**	39 247 409 410 411 413 424 428 431 432 433 438 439 440 441 460 464 465 466 592 593 594
L47	**13**	60 241 243 245 248 249 423 444 445 446 447 461 468
L48	**10**	62 63 204 228 419 422 426 427 429 467
L49	**12**	40 42 237 253 414 418 425 430 457 462 500 501

Level#	Sum	问题一各级最优的基本块排布																
L50	**5**	66	325	415	421	454												
L51	**20**	30	67	73	173	174	176	183	186	238	242	244	246	327	416	417	420	450
		463	502	580														
L52	**9**	43	68	127	175	179	180	184	185	369								
L53	**7**	29	46	177	178	182	206	458										
L54	**8**	71	181	368	442	443	451	604	606									
L55	**2**	128	448															
L56	**11**	5	6	7	8	9	36	45	72	74	342	605						
L57	**8**	10	106	107	122	123	124	126	207									
L58	**20**	70	102	103	104	105	108	109	110	111	112	113	114	115	116	117	118	119
		120	121	125														
L59	**4**	4	69	75	344													
L60	**15**	35	44	47	76	78	326	329	345	348	350	351	353	354	355	393		
L61	**4**	48	328	332	333													
L62	**9**	77	79	80	81	346	347	349	352	586								
L63	**8**	85	99	100	101	130	334	335	336									
L64	**11**	23	82	83	84	129	131	337	339	340	587	590						
L65	**11**	24	25	26	57	87	96	208	338	366	367	591						
L66	**13**	86	89	90	91	93	95	98	209	210	356	357	588	589				
L67	**10**	0	1	2	88	92	94	97	358	359	360							

```
✓ # if(__name__ == "__main__"): ···
···
num_violate_ TCAM = 0
num_violate_ HASH = 0
num_violate_ ALU = 0
num_violate_ QUALIFY = 0
num_control_dependent_violate = 0
num_wr_dependent_violate = 0
num_ww_dependent_violate = 0
num_rw_dependent_violate = 0
```

图 4-6　最优解可行性验证

进一步对芯片流水线各级基本块各资源占用情况进行统计分析（表12），计算每一级各资源（TCAM, HASH, ALU, QUALIFY）的占用率以及所有资源的占用率均值，如图4-7(a) 所示，各级资源占用率均在 40% 到 100% 左右，在考虑依赖约束的前提下占用率较高，资源利用较为高效。分析每种资源在所有级的占用率均值，四种资源分别为 79.4%，53.7%，64.7% 和 88%，TCAM 和 QUALIFY 资源相对较为紧缺，而 HASH 和 ALU 资源相对较为充裕。其中值得注意的是，HASH 资源占用率 100% 的比例较大，所以在下一问中考虑 HASH 资源共享的情况后可能对整体占用级数有较为明显的减少效果。各级各资源的总体占用率均值为 71.5%，资源利用率整体较高，从侧面证明了我们的求解结果的有效性。

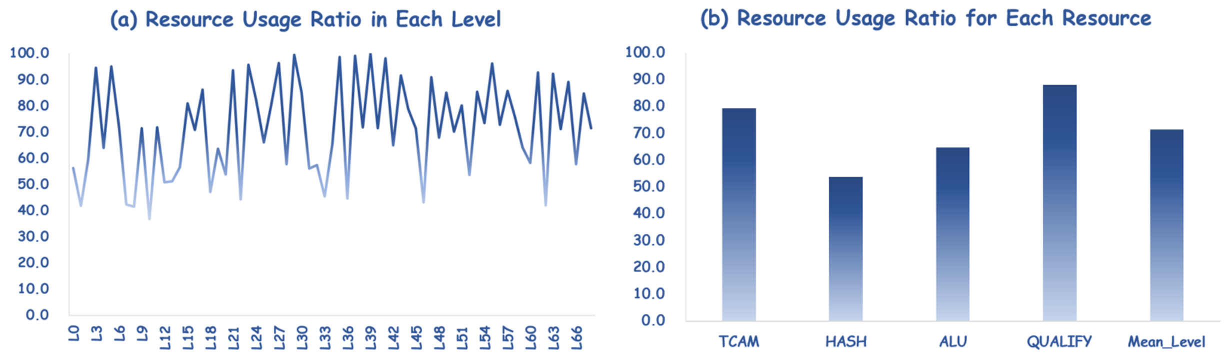


图 4-7　各级资源占用率情况分析

表 12　各流水级基本块资源占用分析

Level	TCAM(%)	HASH(%)	ALU(%)	QUALIFY(%)	Mean_Level(%)
L0	100.0	50.0	0.0	75.0	**56.3**
L1	100.0	0.0	1.8	65.6	**41.9**
L2	100.0	0.0	41.1	96.9	**59.5**
L3	100.0	100.0	87.5	90.6	**94.5**
L4	100.0	0.0	60.7	95.3	**64.0**
L5	100.0	100.0	80.4	100.0	**95.1**
L6	100.0	50.0	64.3	73.4	**71.9**
L7	0.0	100.0	30.4	39.1	**42.4**
L8	0.0	0.0	67.9	98.4	**41.6**
L9	0.0	100.0	87.5	98.4	**71.5**
L10	0.0	0.0	53.6	93.8	**36.9**
L11	0.0	100.0	87.5	100.0	**71.9**

Level	TCAM(%)	HASH(%)	ALU(%)	QUALIFY(%)	Mean_Level(%)
L12	0.0	50.0	64.3	89.1	**50.9**
L13	100.0	0.0	35.7	68.8	**51.1**
L14	100.0	0.0	46.4	79.7	**56.5**
L15	100.0	100.0	44.6	79.7	**81.1**
L16	100.0	50.0	53.6	79.7	**70.8**
L17	100.0	100.0	60.7	84.4	**86.3**
L18	0.0	50.0	57.1	81.2	**47.1**
L19	0.0	100.0	73.2	81.2	**63.6**
L20	0.0	50.0	73.2	92.2	**53.9**
L21	100.0	100.0	80.4	93.8	**93.6**
L22	0.0	0.0	82.1	95.3	**44.4**
L23	100.0	100.0	89.3	93.8	**95.8**
L24	100.0	50.0	85.7	93.8	**82.4**
L25	100.0	0.0	78.6	85.9	**66.1**
L26	100.0	50.0	78.6	92.2	**80.2**
L27	100.0	100.0	85.7	100.0	**96.4**
L28	100.0	0.0	62.5	68.8	**57.8**
L29	100.0	100.0	98.2	100.0	**99.6**
L30	100.0	50.0	91.1	100.0	**85.3**
L31	100.0	0.0	44.6	79.7	**56.1**
L32	100.0	0.0	75.0	54.7	**57.4**
L33	0.0	0.0	82.1	100.0	**45.5**
L34	100.0	0.0	69.6	92.2	**65.5**
L35	100.0	100.0	96.4	98.4	**98.7**
L36	0.0	0.0	78.6	100.0	**44.7**
L37	100.0	100.0	96.4	100.0	**99.1**
L38	100.0	0.0	87.5	100.0	**71.9**
L39	100.0	100.0	100.0	100.0	**100.0**

Level	TCAM(%)	HASH(%)	ALU(%)	QUALIFY(%)	Mean_Level(%)
L40	100.0	0.0	87.5	98.4	**71.5**
L41	100.0	100.0	94.6	98.4	**98.3**
L42	100.0	0.0	66.1	93.8	**65.0**
L43	100.0	100.0	71.4	95.3	**91.7**
L44	100.0	50.0	73.2	92.2	**78.9**
L45	100.0	100.0	5.4	79.7	**71.3**
L46	100.0	0.0	8.9	64.1	**43.3**
L47	100.0	100.0	73.2	90.6	**91.0**
L48	100.0	0.0	83.9	87.5	**67.9**
L49	100.0	100.0	60.7	79.7	**85.1**
L50	100.0	0.0	85.7	95.3	**70.3**
L51	100.0	100.0	41.1	79.7	**80.2**
L52	100.0	0.0	17.9	96.9	**53.7**
L53	100.0	100.0	44.6	96.9	**85.4**
L54	100.0	100.0	0.0	93.8	**73.5**
L55	100.0	100.0	94.6	90.6	**96.3**
L56	100.0	50.0	55.4	85.9	**72.8**
L57	100.0	100.0	55.4	87.5	**85.7**
L58	100.0	100.0	3.6	100.0	**75.9**
L59	0.0	100.0	62.5	93.8	**64.1**
L60	100.0	0.0	64.3	68.8	**58.3**
L61	100.0	100.0	83.9	87.5	**92.9**
L62	0.0	0.0	71.4	96.9	**42.1**
L63	100.0	100.0	78.6	90.6	**92.3**
L64	100.0	0.0	91.1	93.8	**71.2**
L65	100.0	100.0	67.9	89.1	**89.3**
L66	100.0	0.0	71.4	59.4	**57.7**
L67	100.0	100.0	50.0	89.1	**84.8**

Level	TCAM(%)	HASH(%)	ALU(%)	QUALIFY(%)	Mean_Level(%)
Average_Resource	79.4	53.7	64.7	88.0	71.5

4.7.2 算法有效性和时间复杂度分析

基于问题一的求解结果，我们对本文提出的**目标松弛诱导定界和变量固定驱动的分支切割算法**的有效性以及时间复杂度进行分析，以直接调用求解器作为比对基准，具体结果见表13。

表 13 算法有效性及复杂度比对分析 (直接调用求解器 v.s. 精确加速算法)

算法	直调调用求解器	目标界限诱导变量固定算法
变量总数	369054	41412
约束总数	129830	125864
探索节点数量	1997	16783
求解时间	**90939 秒**	**5667 秒**
最优解	**未找到可行解，下界 61**	**最优解 68**
算法复杂度	$\mathcal{O}(\mathrm{IP})$	$\log_2(\mathrm{UB}_0 - \mathrm{LB}_0)\mathcal{O}(\text{OR-IP})^*$

*: UB_0 和 LB_0 分别表示初始的上界和下界

算法有效性：本文设计的精确加速算法**在 1.5 小时左右得到全局最优解 68**，结果可行性由可行性检验保证，最优性由目标松弛诱导定界保证。与之相对的，若直接调用求解器求解，**25 小时仍未找到任何可行解，单纯形法的下界只更新到了 61**。针对芯片资源排布问题，求解结果的精确性非常重要，本文的精确加速算法可以在较短时间内获得最优的资源排布方案，有效保证芯片资源的最高效利用，有效节约大量资源成本。

算法时间复杂度：芯片资源排布整数规划问题是一个 NP-hard 问题，无法在多项式时间内求解。从求解时间来看，该模型的求解复杂度可以通过直接调用求解器看出，原始模型变量总数为 369054，约束总数为 129830，在**90939 秒**内求解器调用的分支切割算法只能探索 1997 个节点，并且没有找到任何可行解。而本文设计的精确加速算法能够将变量总数削减到 41412 个，有效降低模型求解的时间复杂度，对模型求解进行显著加速，在**5667 秒**内就探索了 16783 个节点，并直接找到全局最优解。

从算法复杂度来看，虽然原始的整数规划问题 (Integer Programming, IP) 和目标松弛诱导问题 (Objective Relaxation - Integer Programming, OR-IP) 均为 NP-hard 问题，但后者的计算复杂度 $\mathcal{O}(\text{OR-IP})$ 显著低于前者 $\mathcal{O}(\mathrm{IP})$。在我们的目标松弛诱导定界算法中，初始的目标函数界限为 $[0, |\mathcal{B}|]$，如图4-8所示。将通过目标松弛诱导定界算法找到的第一个不可行解设为第一个下界 **LB_0**，找到的第一个可行解设定为第一个上界 **UB_0**。接下来采用二分法将 **(LB_0+UB_0)/2** 设定为新的上界迭代计算，若得到可行解

则更新 **UB_1 = (LB_0+UB_0)/2**，反之若得到不可行解则更新 **LB_1+1 = (LB_0+UB_0)/2**，反复迭代直到上下界重合，得到最优解 **UB=LB**。于是，我们的算法复杂度可以表示为$\underline{\log_2(\mathbf{UB\text{-}LB})\cdot \mathcal{O}(\mathbf{OR\text{-}IP})}$，依然显著低于直接调用求解器的复杂度$\underline{\mathcal{O}(\mathbf{IP})}$。

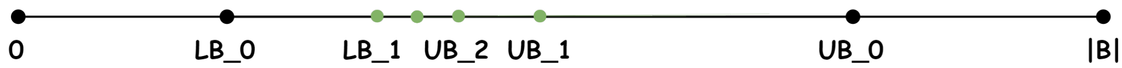


图 4-8　二分法诱导定界示意图

5. 问题二模型建立与求解

5.1 问题二分析

在问题一基础上，问题二增加对不同执行流程上基本块资源共享的考虑。在同一流水级中，不同执行流程上的基本块可以共享 HASH 资源和 ALU 资源，其中任意一个基本块的资源要求均不超过每级资源限制即可。据此，对问题一中 HASH 资源和 ALU 资源的资源上限约束以及 HASH 资源的折叠资源约束进行更新。针对流水线中每一级，不再对所有基本块资源进行约束，而是只对在同一条执行流程上的基本块进行限制。这样相比问题一，问题二对资源约束进行了松弛，可能可以得到更少的占用流水线级数。

对问题二进行建模要的关键问题是基于 P4 流程图进行同执行流程基本块组的识别。首先，使用本文提出的 **可行路判别算法 (FeasiblePath)** 识别同执行流程基本块对（即判断任意两个基本块对是否是可达的），并基于此构建出**基本块邻接网络**。通过寻找基本块邻接网络中的所有路径，就可以找到所有的包含同一执行流程的不同的基本块组。但是，基本块对邻接网络中的所有路径是指数级爆炸的，并且很多路径之间会有重合关系，这就导致很多路径其实是冗余的。为了最大限度地降低复杂度，我们提出了一个 **网络缩减算法** 删除基本块邻接网络中可以被替代的边。经过网络缩减算法处理的之后，剩余的网络中任意两点之间最多只有一条不同的路径，这些路径是之间互相不存在替代路径，并且网络中的所有路径的数量可以降低成千上万倍，算法运行速度也得到了显著提升。

基于上述算法，我们根据问题 2 的题目要求构建了**考虑资源共享的资源排布整数规划模型**，并设计了高效的**精确求解算法** (**松弛和约束生成-可行性证明定界算法**) 进行模型求解。

图4-4给出了整个算法的框架图，图中展示了各个组成部分。这里对整体框架做简要介绍。

1. 首先，构建问题 2 的整数规划模型；根据初始的目标值的界限，并行多个模型；
2. 然后将资源约束松弛；
3. 之后将模型的目标函数设置为 0，也就是执行目标松弛；
4. 将目标松弛后的模型输入给分支切割算法求解；
5. 若分支切割算法获得一个整数解，则进行可行性检验；
6. 若该解满足所有的资源约束，则得到一个全局的上界，此时执行 **目标界限诱导的变量固定算法** 对变量进行固定，得到子整数规划 (Sub-IP)；
7. 更新全局的模型为 Sub-IP，然后继续执行**分支切割算法**；
8. 若该解不满足所有的资源约束，则执行**资源约束生成算法**，首先，根据建立好的基本块邻接网络，执行网络缩减算法，并生成所有在统一执行流程中的基本块组。根据以上信息，生成资源约束，并加入到模型中，更新模型。更新后的模型继续回到分支切割算法。
9. 若分支切割算法无解，则执行**可行性证明定界**，根据新的界限构建新的并行的整数规划模型。

10. 当全局上界等于全局上界时，算法终止，得到最优解。

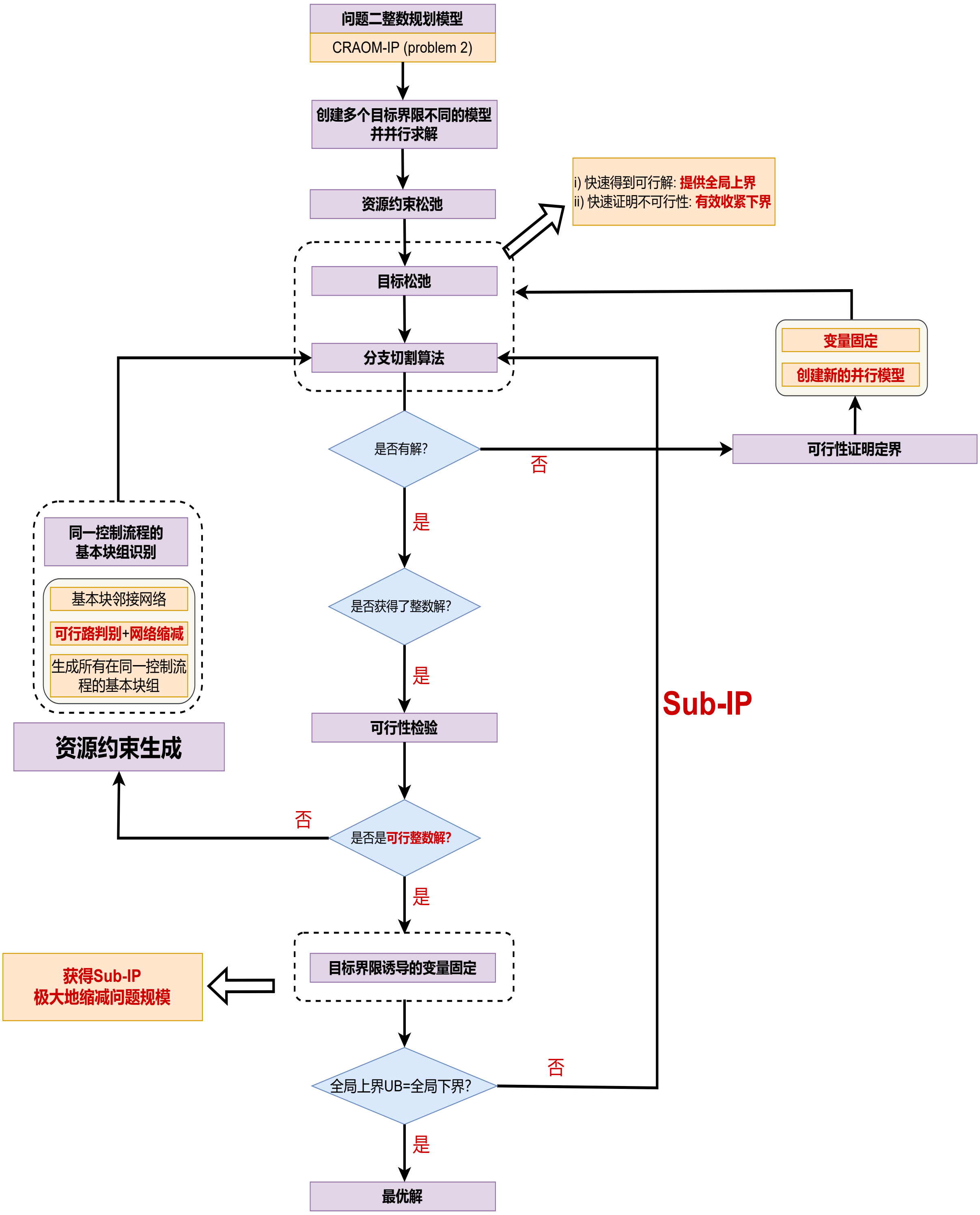


图 5-1　问题二求解算法框架：松弛和约束生成-可行性证明定界算法

5.2 同一执行流程的基本块对识别：可行路判别算法

针对流水线中的每一级，我们需要先判断哪些基本块属于同一执行流程，生成所有同执行流程的基本块组合，再针对每一个组合去建立各自的资源约束。为了识别出

所有同执行流程的基本块组合，本文提出了一种可行路判别算法来判断二者是否可达。其主要思想是：若两个基本块之间存在可行的路径，则二者就是在同一执行流程中。

我们具体的做法是；对流水线的每一级，根据各基本块在 p4 程序流程图中的可达关系，构建属于该流水级基本块的邻接网络。邻接网络的构建流程如图5-2所示。而后，判断一对基本块对是否是在同一执行流程中，等价于判断二者在邻接网络中否是可达的。

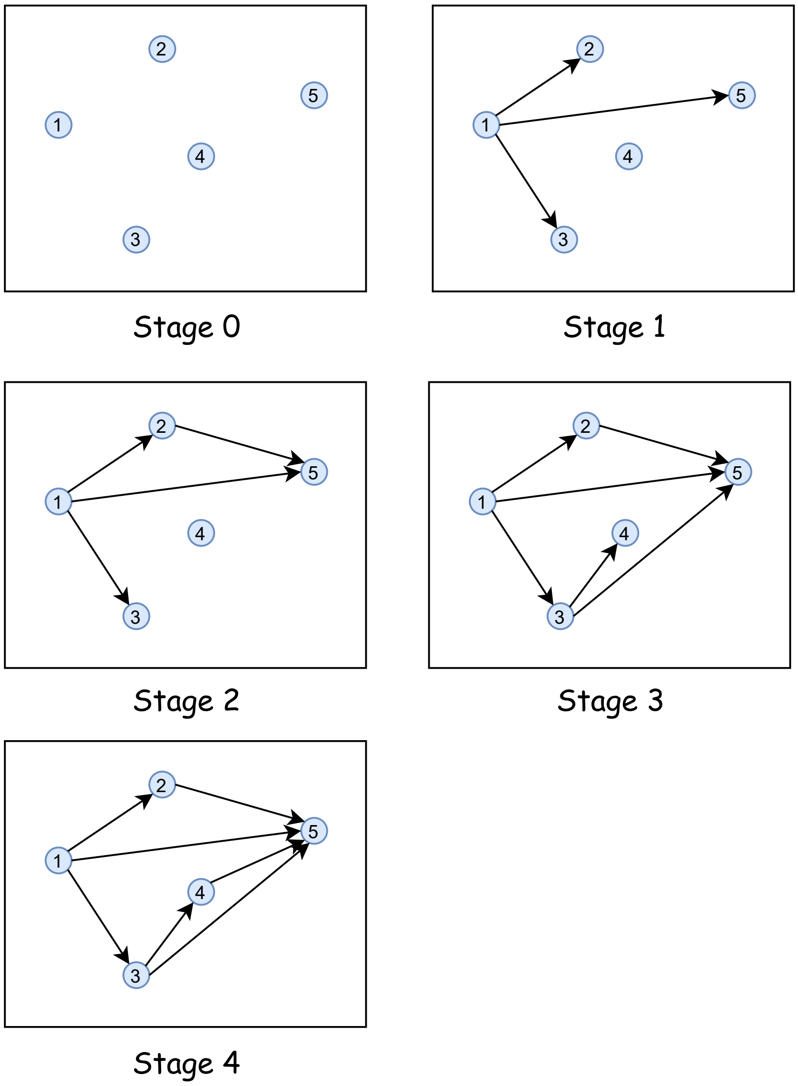


图 5-2　可行路生成算法示例

我们以图5-3为例进行说明如何通过可行路判断同执行流程。假设 [1, 2, 3, 4, 5] 同属一个流水级。我们先根据这些基本块在 p4 流程图中的可达关系，按5-2展示的方法构建该流水级的邻接网络。再针对每一对基本块，使用最短路算法获得可达性，从而判断各基本块否属于同一流程。

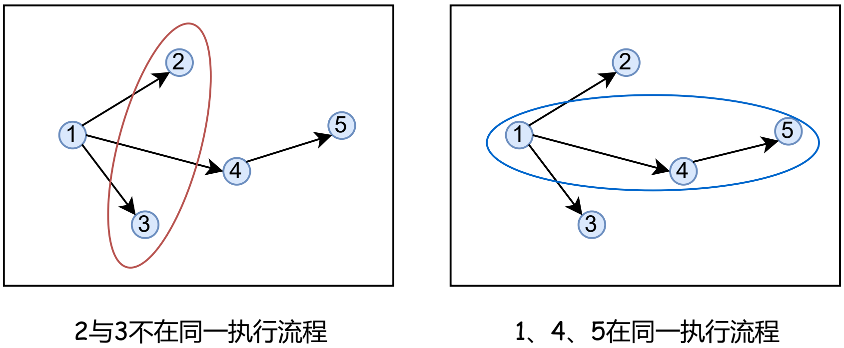


图 5-3　通过是否存在可行路判断是否为同一执行流程

下面我们给出识别每个流水级中同一执行流程基本块对的算法流程（算法3）：

Algorithm 3 同一执行流程基本块对识别算法

Require: 基本块集合 $\mathcal{B}$，邻居矩阵 $\mathcal{A}$

Ensure: 同执行流程基本块对字典 $\mathcal{P}$

```
for 每个基本块 i ∈ B do
    for 每个基本块 j ∈ B do
        if i ≠ j then
            feaPathExist ← FeasiblePath(G, i, j) (i 到 j 之间是否有可行路)
            if feaPathExist == 1 then
                P ← P ∪ {(i, j) : 1}
            end if
        end if
    end for
end for
return P
```

在上述的算法3中，其中一步操作需要识别两个基本块 s 和 t 之间是否有可行的路径。该问题可以被建模为下面的约束规划模型。在该模型中，我们用 $x_{ij} \in 0, 1, \forall i, j \in \mathcal{B}$ 表示在网络中弧 (i, j) 是否会被选中。由此，识别可行路的模型可以写成如下的形式。

$$\min \quad 0 \tag{59}$$

$$s.t. \quad \sum_{j \in \mathcal{B}} x_{ij} - \sum_{j \in \mathcal{B}} x_{ji} = \begin{cases} 1, & \text{if } i = s, \\ 0, & i \in \mathcal{B} \backslash \{s, t\}, \\ -1, & \text{if } i = t, \end{cases} \tag{60}$$

$$x_{ij} \in \{0, 1\}, \qquad \forall (i, j) \in \mathcal{A}. \tag{61}$$

模型 (59)-(61) 可以使用改进的 Dijkstra 算法快速求解，或者使用深度优先算法快速

求解。下面我们给出可行路判别算法（算法4）和基于深度优先的 Dijkstra 算法 (算法5) 的伪代码流程。

Algorithm 4 可行路判别算法 (`FeasiblePath`)

Require: 网络图 $\mathcal{G}$，起点 s，终点 t

Ensure: 起点和终点是否存在可行路径 `feaPathExist`

`feaPathExist` $\leftarrow$ `Dijkstra` $(\mathcal{G}, s, t)$

return `feaPathExist`

Algorithm 5 基于深度优先的 Dijkstra 算法

Require: 有向图网络 $\mathcal{G}$

Ensure: 起始点 s 到所有节点 $b \in \mathcal{B}$ 的可达路径

for 点 $i \in$ Graph **do**
    到起始点最短距离 $d_i \leftarrow \infty$
    i 的前一个节点 $p_i \leftarrow$ undefined
end for
$d_s \leftarrow 0$
$G \leftarrow$ Graph 中的所有点的集合
while G 非空 **do**
    $b \leftarrow G$ 中 d 最小的点
    $G \leftarrow G \backslash \{b\}$
    for b 的每个邻居节点 i **do**
        $d_{temp} \leftarrow d_b + c_{b,i}$ ($c_{b,i}$ 表示点 b, i 之间的距离)
        if $d_{temp} < d_i$ **then**
            $d_i \leftarrow d_{temp}$
            $p_i \leftarrow b$
        end if
    end for
end while
return p

5.3 网络缩减算法：缩减基本块邻接网络的规模

5.3.1 算法瓶颈

我们对算法3进行了编程实现，尝试对每个流水级中同一执行流程基本块组进行识别。为了对可行路进行快速判别，我们在网络中引入虚拟起点和虚拟终点，然后生成该网络中的所有可行路径，就可以得到所有的在同一控制流程中的基本块组。

然而经过测试，这种方法非常低效，是因为有很多基本块组之间是具有**可替代关系**的，是重复的。例如，若流水级 k 排布了基本块 0, 1, 2, 4, 586, 587, 589, 590, 591，假设以

基本块 4 为起点，基本块 2 为终点，则可得到如下的所有可行路径。

$$\text{基本块组 } 1: [4, 0, 1, 2] \tag{62}$$

$$\text{基本块组 } 2: [4, 0, 2] \tag{63}$$

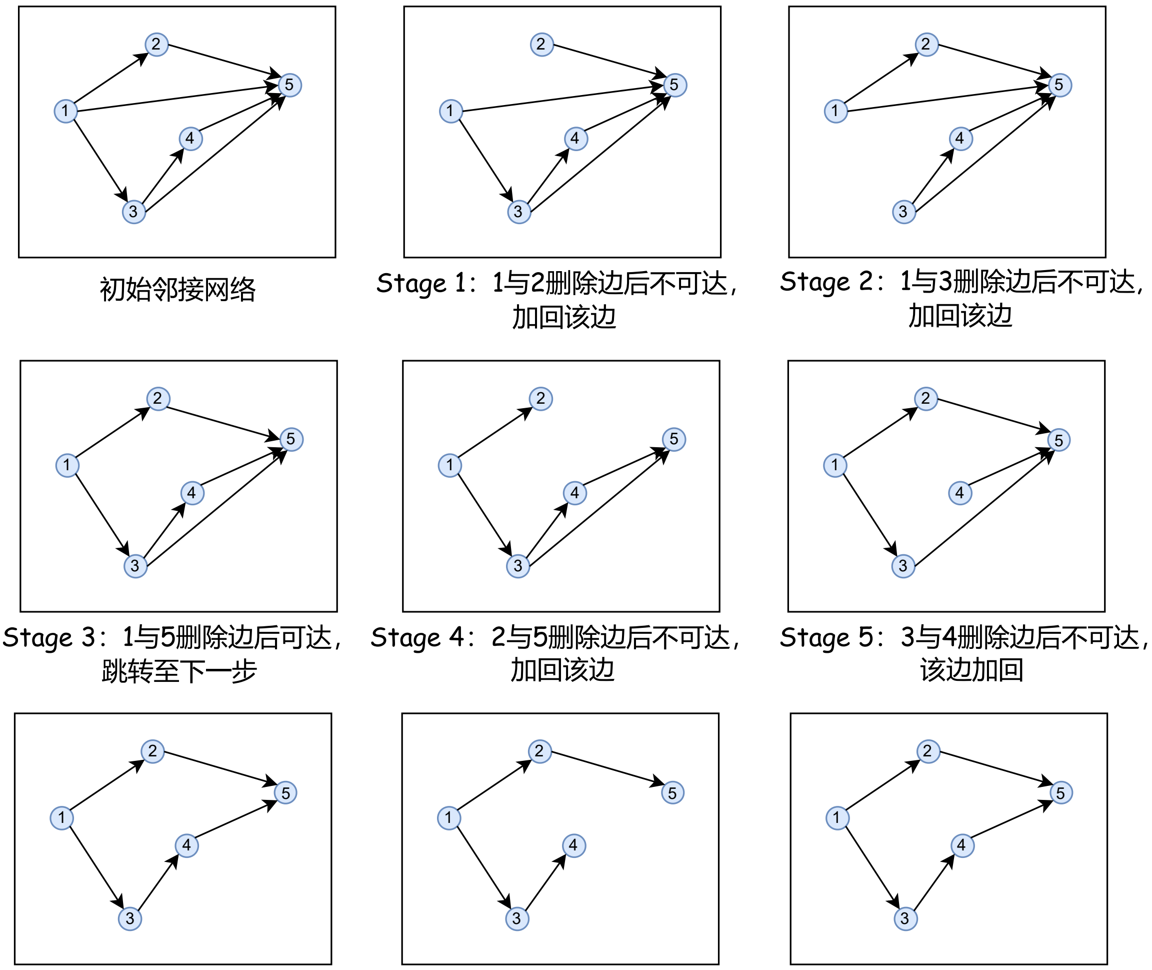
$$\text{基本块组 } 3: [4, 586, 587, 590, 591, 589, 0, 2] \tag{64}$$

观察到上述 3 组基本块中，第 2 组中的基本块完全包含在第 1 组中，因此第 2 组是冗余的。冗余的组若不删除，就会导致约束数量急剧增多，不利于求解。一个简单的删除冗余基本块组的想法是：穷举所有的可行路，然后识别冗余的路径。但是这种做法是不可行的。首先，找到所有的可行路径会非常耗时，我们再测试的过程中，发现一些情况运行半小时以上都不能选找到所有的可行路径。其次，识别是否冗余，也会花费大量的时间。

5.3.2 算法设计

为了解决约束冗余的问题，我们提出了一个巧妙的办法：在对每个流水级的基本块生成邻接网络后，我们先尝试对网络图进行缩减，删除那些可以被替代的边，保证剩余的网络图中，2 个节点之间的可达路径至多只有 1 条。如图5-4展示的网络缩减算法示例，网络缩减的主要思路是：对网络图 $\mathcal{G}$ 中的每一条边 (i, j)，首先将其从 $\mathcal{G}$ 中删除，然后设置 i 为起点，j 为终点，调用算法 4和算法5 判断是否有可行路径，若有，则继续检查其他边 (也就是弧 (i, j) 是可以被可行路径替代的)，若无可行解，则将弧 (i, j) 加回。我们设计的网络缩减算法可以保证以下两点：

1. 不破坏原图中任意两点之间的可达性；
2. 缩减后的图中，任意两点之间只有一条可行路径。



初始邻接网络

Stage 1：1与2删除边后不可达，加回该边

Stage 2：1与3删除边后不可达，加回该边

Stage 3：1与5删除边后可达，跳转至下一步

Stage 4：2与5删除边后不可达，加回该边

Stage 5：3与4删除边后不可达，该边加回

Stage 6：3与5删除边后可达，跳转至下一步

Stage 7：4与5删除边后不可达，加回该边

Stage 8：网络缩减算法完成

图 5-4　网络缩减算法示例

下面我们给出网络缩减算法的伪代码流程。

Algorithm 6 网络缩减算法

Require: 初始基本块邻接网络图 $\mathcal{G}$

Ensure: 缩减后的基本块邻接网络图 $\mathcal{G}$

for 每条边 $(i, j) \in \mathcal{G}$ **do**

　删除边 (i, j): $\mathcal{G} \leftarrow \mathcal{G}.\mathtt{removeEdge}(i, j)$

　检查 i 和 j 之间是否有可行路径: $\mathtt{feaPathExist} \leftarrow \mathtt{FeasiblePath}(\mathcal{G}, i, j)$

　if $\mathtt{feaPathExist}$ == True **then**

　　continue

　else if $\mathtt{feaPathExist}$ == False **then**

　　将边 (i, j) 加回： $\mathcal{G} \leftarrow \mathcal{G}.\mathtt{addEdge}(i, j)$

　end if

return $\mathcal{G}$

5.3.3 算法结果与复杂性分析

为体现我们提出的网络缩减算法的效率优势，我们以所有基本块编号的前 100 个基本块为例，我们线松弛了约束70和约束72，并获得了如下的求解结果：

流水级 0 | 分配的基本块：5 6 7 8 13 14 15 16 17 18 19 20 21 23 24 25 26 31 32 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 57 58 59 60 61 63 65 66 70 74 82 83 84 86 91 92 93 94
流水级 1 | 分配的基本块：4 10 29 30 64 67 68 69 71 72 73 75 76 78 90 96
流水级 2 | 分配的基本块：9 11 12 22 62 77 79 80 81 88 95 97
流水级 3 | 分配的基本块：0 1 2 3 27 28 33 34 35 56 85 87 89 98 99

我们使用该求解结果中最长的流水级 0。

首先我们不进行网络缩减，而直接运行同一执行流程基本块对识别算法（算法3）。程序在提取所有在同一执行流程的基本块组的过程中，出现了非常严重的指数爆炸现象。具体来说，我们设置了运行时间为 30 分钟，然而程序依旧不能输出所有在同一控制流程的基本块组。

随后我们在调用了网络缩减算法（算法6）。我们先执行算法6，再让算法3基于缩减后的图去识别同一控制流程的基本块组。如图5-5所示，新算法在**2.6 秒**后即成功识别了流水级 0 中所有同一执行流程的基本块组，可见网络缩减算法极强的算法加速能力。

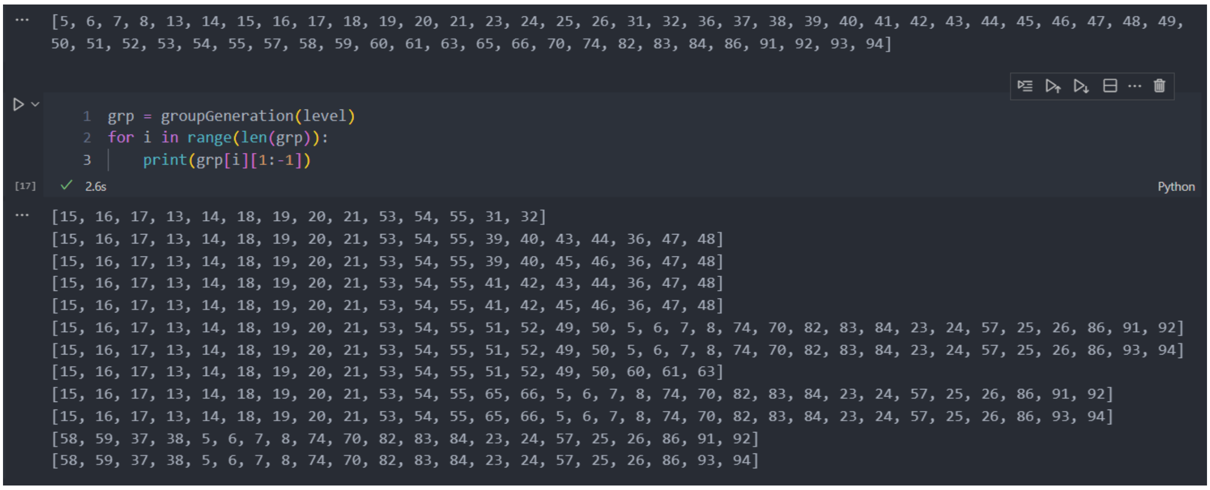


图 5-5 网络缩减算法的加速效果

5.4 问题二数学模型：考虑资源共享的资源排布整数规划模型

问题二的数学模型中，其他约束和问题一相同，资源约束方面却发生了明显的变化。因此本节只对资源约束的刻画进行详细解释。

资源约束的变化来源于：不同执行流程上的资源是可以共享的。用 Ω_k 表示被分配到第 k 个流水级上的所有在同一流程上的基本块组的集合。下面举一个例子来说明本题中资源约束的刻画。

仍以前面小节中的例子为例，我们在使用网络缩减算法处理完图之后，找到了第 2 个流水级中在同一执行流程中的基本块组如下

$$\Omega_2 = \{[4,0,1,2],[4,586,587,590,591,589,0,2]\}$$

根据题意，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，若以第 1 组 [4, 0, 1, 2] 为例，该约束可以写为（注意流水级 ID 为 2）

$$x_{4,2}d_{4,HASH} + x_{0,2}d_{0,HASH} + x_{1,2}d_{1,HASH} + x_{2,2}d_{2,HASH} \leqslant 2.$$

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，对应的约束为

$$x_{4,2}d_{4,ALU} + x_{0,2}d_{0,ALU} + x_{1,2}d_{1,ALU} + x_{2,2}d_{2,ALU} \leqslant 56.$$

下面给出上述资源约束的一般形式。基于上述描述，可以将约束（43）改下为如下的形式

$$\sum_{b\in S} x_{bk}d_{br} \leqslant U_r, \qquad \forall k\in\mathcal{K}, r\in\{HASH, ALU\}, S\in\Omega. \tag{65}$$

公式 (70) 表示同一级中，不在同一控制流程中的基本块的资源占用量小于等于上限，其中 $U_{HASH}=2, U_{ALU}=56$。

下面是折叠层级的 TCAM 的约束。表示折叠的两级，对于 TCAM 资源约束不变 ($Q_{TCAM}=1$)。

$$\sum_{b\in\mathcal{B}} (d_{b,TCAM}x_{bk} + d_{b,TCAM}x_{b,k+16}) \leqslant Q_{TCAM}, \qquad \forall k\in\{0,1,\cdots,15\}. \tag{66}$$

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($Q_{TCAM}=3$)

$$\sum_{b\in\mathcal{S}_1} d_{b,HASH}x_{bk} + \sum_{b\in\mathcal{S}_2} d_{b,HASH}x_{b,k+16} \leqslant Q_{HASH}, \quad \forall k\in\{0,1,\cdots,15\}, S_1, S_2\in\Omega. \tag{67}$$

因此，问题 2 的完整的数学模型为

$$\min \sum_{k\in\mathcal{K}} y_k \tag{68}$$

$$s.t. \quad 约束(39)\sim(42),(45)\sim(50). \tag{69}$$

$$\sum_{b\in S} x_{bk}d_{br} \leqslant U_r, \qquad \forall k\in\mathcal{K}, r\in\{HASH, ALU\}, S\in\Omega. \tag{70}$$

$$\sum_{b\in\mathcal{B}} (d_{b,TCAM}x_{bk} + d_{b,TCAM}x_{b,k+16}) \leqslant Q_{TCAM}, \qquad \forall k\in\{0,1,\cdots,15\}. \tag{71}$$

$$\sum_{b\in\mathcal{S}_1} d_{b,HASH}x_{bk} + \sum_{b\in\mathcal{S}_2} d_{b,HASH}x_{b,k+16} \leqslant Q_{HASH}, \quad \forall k\in\{0,1,\cdots,15\}, S_1, S_2\in\Omega. \tag{72}$$

注意到约束 (70) 和约束 (72) 数量庞大，其数量为 $|\Omega|$。Ω 是所有在同一控制流程上的基本块的集合的集合。经过测算，最坏情况下，Ω 中至少包含上百万个元素。因此，穷举这些元素是很低效的。为了提高效率，我们提出了一种**松弛和约束生成-目标松弛诱导定界**的精确算法框架来高效求解该问题。

5.5 精确求解算法设计：松弛和约束生成-可行性证明定界算法

松弛和约束生成-可行性证明定界算法的主要思路是：首先将约束 (70) 和约束 (72) 松弛，然后使用第一问中提出的目标松弛诱导定界和变量固定驱动的分支切割算法求解模型 (68)-(70)，若得到整数可行解 $(\bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{z}})$，则检查该整数可行解是否违背约束 (70) 和约束 (72)，若违背，则调用算法3、算法 4、算法 6找到每一个流水级中的在同一执行流程上的基本块组的集合。然后调用约束生成算法，生成约束 (70) 和 (72)，以删去该不可行解。注意到该过程中，约束 (70) 和 (72) 中只有被违反的部分才会被加入到模型中，因此该算法非常高效。

下面是**松弛和约束生成-可行性证明定界算法** 算法的伪代码。

Algorithm 7 松弛和约束生成-可行性证明定界算法

Require: 松弛后的模型 (68)-(69) 和 (71)

Ensure: 最优解 $(\bar{\mathbf{x}}^*, \bar{\mathbf{y}}^*, \bar{\mathbf{z}}^*)$ 和最优值 Z^*

初始化全局上界 $Z_l \leftarrow Z_{LPrelaxation}$，全局上界 $Z_u \leftarrow |\mathcal{K}|$, 最优容差 $\epsilon \leftarrow 0.00001$
while $Z_u - Z_l > \epsilon$ **do**
  调用目标松弛诱导定界和变量固定驱动的分支切割算法求解模型 (68)-(69) 和 (71)
  if 得到整数解 $(\bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{z}})$ **then**
    /* 检查 $(\bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{z}})$ 的可行性 */
    for 每一个流水级 $k\in\mathcal{K}$ **do**
      $\Omega_k \leftarrow$ 调用算法3、4、6 识别同一控制流程的基本块组的集合
      if $\Omega_k = \varnothing$ **then**
        /* **可行性证明定界和变量固定** */
        更新全局的上界：$Z_u \leftarrow \min\{Z_u, Z(\bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{z}})\}$
        更新最优值 $Z^* \leftarrow \min\{Z_u, Z^*\}$，更新最优解
        变量固定：对符合条件的决策变量执行变量值的固定

else

/* **资源约束生成** */

约束生成：生成约束 (70) 和约束 (71)，并加入到 (68)-(69) 和 (71) 中

end if

end for

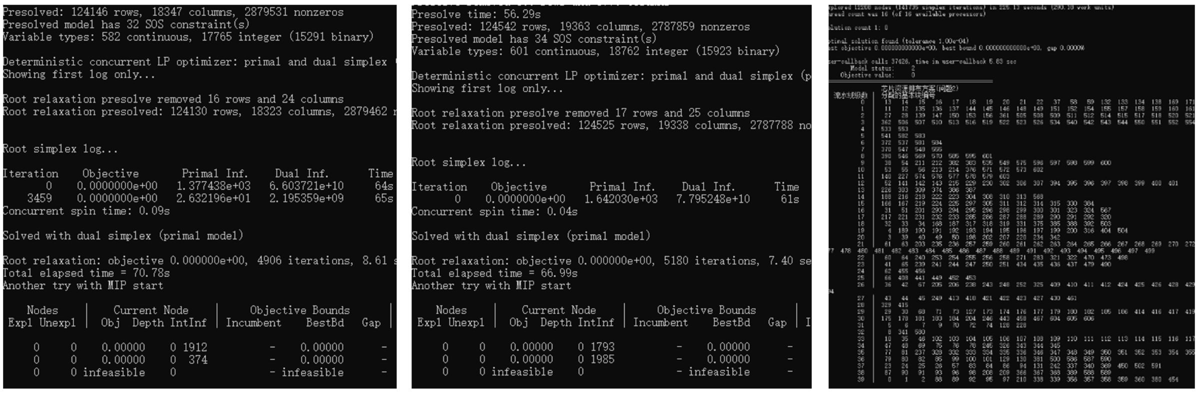
end if

end while

return 最优解 $(\bar{x}^*, \bar{y}^*, \bar{z}^*)$ 和最优值 Z^* =0

5.6 问题二求解结果

5.6.1 结果与分析



(a) Z^u = 37 Infeasible　(b) Z^u = 38 Infeasible　(c) Z^u = 40 Feasible

图 5-6　目标松弛诱导定界结果 ($Z^u = 39$ Optimal)

针对问题二，本文设计松弛和约束生成-可行性证明定界算法，求解考虑资源共享的资源排布整数规划模型。定界过程中获得的上下界 (不可行解 37, 38 和可行解 40) 如图5-6所示，最优解求解结果如图5-7所示，表14中展示了问题二中各流水级最优的基本块排布方案和各流水级所排布的基本块数量。平均每一级分配 16 个基本块，第 20 级被分配的基本块最多，为 37 个；第 5 级和第 7 级被分配的基本块数量最少，仅为 1 个。总流水级数最优解为39。问题二求解结果详见附件问题二结果.xlsx。

表 14　问题二各级基本块排布求解结果

总流水级数最优解：39

Level#	Sum	问题二各级最优的基本块排布																
L0	**24**	13	14	15	16	17	18	19	20	21	22	37	58	59	132	133	134	138
		169	171	363	364	365	373	391										
L1	**26**	11	12	135	136	144	145	146	148	149	151	152	154	155	157	158	159	160

Level#	Sum	问题二各级最优的基本块排布																
		161	162	163	164	165	170	172	376	393								
L2	**33**	27	28	137	147	150	153	156	361	377	505	508	509	511	512	514	515	517
		518	520	521	524	525	527	528	529	530	531	532	533	536	538	539	545	
L3	**27**	362	506	507	510	513	516	519	522	523	526	540	542	543	544	552	553	554
		556	557	558	559	560	561	562	563	564	565							
L4	**3**	378	534	566														
L5	**1**	555																
L6	**3**	537	547	548														
L7	**1**	546																
L8	**7**	390	535	541	581	582	583	584										
L9	**15**	54	211	212	549	569	570	571	575	576	577	578	579	585	595	596		
L10	**10**	53	55	56	213	214	572	597	598	599	600							
L11	**8**	38	140	227	573	574	601	602	603									
L12	**30**	141	142	143	215	226	229	302	306	307	312	313	370	371	372	374	375	379
		382	383	384	386	392	394	395	396	397	398	399	400	401				
L13	**21**	51	52	139	166	167	188	218	219	221	222	223	224	225	230	303	305	308
		309	310	311	385													
L14	**10**	201	293	297	298	299	304	314	315	323	330							
L15	**7**	216	231	294	295	296	300	301										
L16	**12**	187	232	233	285	286	288	289	290	291	317	320	324					
L17	**11**	33	34	168	287	292	318	319	331	550	551	568						
L18	**17**	3	32	189	190	191	192	193	194	196	199	200	205	206	207	316	503	504
L19	**19**	36	39	40	41	42	44	49	195	197	198	217	220	234	325	327	342	402
		404	405															
L20	**37**	45	50	60	61	63	202	203	235	260	261	263	265	273	274	276	277	279
		280	284	387	403	406	407	469	475	476	478	480	482	483	488	489	491	492
		494	495	499														
L21	**18**	43	64	257	258	267	268	270	271	272	282	283	321	470	472	473	485	486

Level#	Sum	问题二各级最优的基本块排布																
		498																
L22	**18**	236	239	259	262	264	266	269	275	278	281	322	326	329	388	434	477	479
		567																
L23	**18**	62	204	228	240	251	252	254	435	456	471	474	481	484	487	490	493	496
		497																
L24	**16**	35	238	247	248	250	253	381	389	408	412	436	437	449	453	455	592	
L25	**27**	46	47	66	255	256	409	410	411	421	424	426	428	429	430	431	432	433
		438	439	440	441	444	452	457	463	593	594							
L26	**17**	48	65	67	237	328	332	380	413	418	419	422	423	425	427	454	459	468
L27	**7**	245	333	334	415	445	451	464										
L28	**15**	29	173	246	249	414	416	417	420	446	460	461	462	465	466	501		
L29	**23**	30	68	71	73	174	175	176	177	178	180	181	183	184	185	186	242	339
		442	443	447	458	467	606											
L30	**9**	9	72	127	128	179	182	448	604	605								
L31	**6**	5	6	7	8	70	74											
L32	**22**	10	31	102	103	104	105	106	107	108	109	111	112	113	115	116	117	119
		120	121	123	124	126												
L33	**13**	69	75	76	78	110	114	118	122	125	241	341	344	345				
L34	**15**	4	77	79	81	343	346	347	348	349	350	351	352	353	354	355		
L35	**15**	80	82	85	99	100	101	129	130	243	337	450	500	586	587	590		
L36	**14**	23	24	25	26	57	83	84	86	131	335	336	369	502	591			
L37	**17**	87	90	91	92	93	94	96	208	209	210	340	366	367	368	580	588	589
L38	**15**	0	1	2	88	89	95	97	98	244	338	356	357	358	359	360		

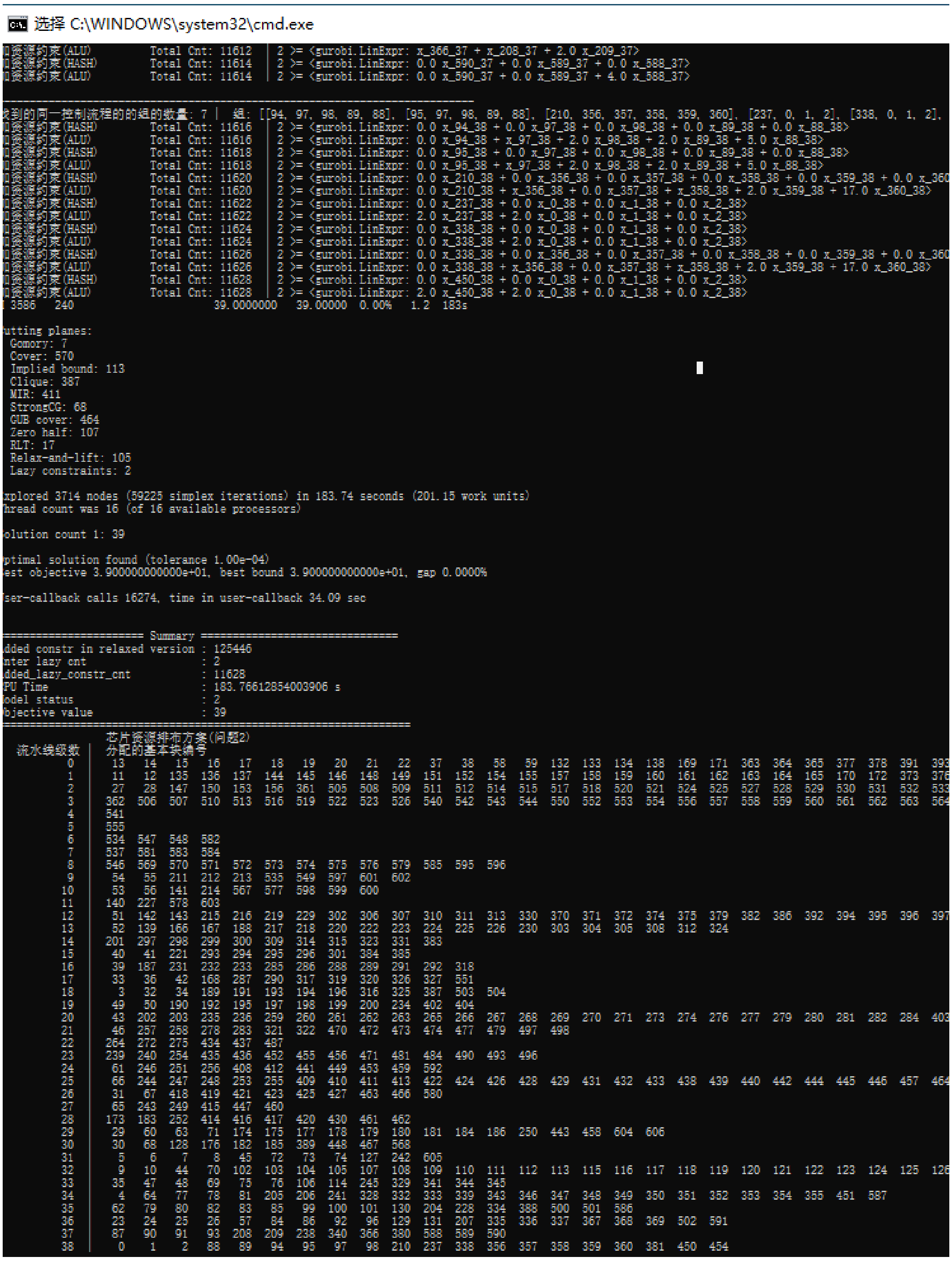


图 5-7　松弛和约束生成-可行性证明定界算法最优解结果 ($Z^u = 39$)

5.6.2 算法有效性和时间复杂度分析

基于问题二的求解结果，我们对本文提出的**松弛和约束生成-可行性证明定界算法**的有效性以及时间复杂度进行分析，以直接求解模型作为比对基准，具体结果见表15。

表 15　有效性及复杂度比对分析 (直接求解 v.s. 松弛和约束生成-可行性证明定界)

算法	直接求解模型	松弛和约束生成-可行性证明定界算法
约束总数（除资源约束外）	125446	125446
资源约束总数	$\mathcal{O}(2^{607})$	11628
求解时间	∞	183 秒
最优解	None	最优解 39

算法有效性： 本文设计的精确加速算法**在 183 秒得到全局最优解 39**，结果可行性由可行性检验保证，最优性由目标松弛诱导定界保证。与之相对的，若直接对模型求解，求解时间随问题规模指数级爆炸，**最坏情况的求解时间为无穷大**。松弛和约束生成-可行性证明定界算法针对资源共享的考虑进行约束生成，在极短的时间内获得最优的调度方案，这证明了算法极高的有效性。

算法时间复杂度： 若直接求解模型，要考虑的资源约束总数为$\mathcal{O}(2^{607})$，而通过本文设计的松弛和约束生成-可行性证明定界算法，最终模型添加的资源约束总数为**11628**。对资源约束总数的显著削减高效降低了本文算法的计算时间复杂度，使得直接求解时间极有可能为无穷大的问题在 183 秒内得到了全局最优解。

6. 模型评价

本文通过对 PISA 架构芯片资源排布问题进行了深入研究，在深入理解题意的基础上，紧扣题目要求，建立能够满足控制依赖约束、数据依赖约束与资源约束的数学规划模型。我们针对问题一与问题二分别设计了各种类型的精确算法与加速策略，在保证最优性的前提下，显著提升了模型的求解效率，并成功获得了问题一与问题二的全局最优解。根据我们提供的排布方案，问题一只需 68 个流水级，问题二只需 39 个流水级即可满足题目的所有约束条件。我们结合算法复杂性分析与有效性分析，说明了直接求解原模型的难度，以及我们设计加速算法的必要性，并为我们模型获得的解提供了最优性证明。

6.1 模型的优点

1. 虽然本文的问题是一个 NP-Hard 问题，但是通过各种加速方法，我们仍旧基于精确的方法成功快速获得了该问题的全局最优解。
2. 对于问题一，我们结合最大流理论设计多项式时间算法，成功解决了各种依赖关系的获取与识别。
3. 此外，我们设计了目标松弛技巧，快速获得可行解或者证明不可行性，从而显著地加速了上界和下界的收敛，并且为变量固定提供了有力的支撑。这些精确加速技巧使得算法在 1.5 小时就获得了全局最优解，而直接调用 Gurobi 求解原模型，则 25 小

时内仍无法获得任何可行解。

4. 对于问题二，关键的难点在于枚举出所有同一执行流程的基本块组，我们设计了网络缩减算法减小邻接网络的规模，从而将识别同一执行流程的基本块组的效率提升了 1000 倍以上。
5. 我们针对问题二的约束爆炸的挑战，设计了一种松弛和约束生成算法，以规避对资源约束的穷举。数值实验表明，松弛和约束生成算法仅仅会将有效的资源约束识别并添加到模型中，同时忽略冗余的资源约束。这种操作也使得问题求解效率大大提升，求解问题 2 仅需 183 秒即可获得全局最优解。

6.2 模型的缺点

我们设计的算法针对目标函数为最小化所使用的流水级非常高效，但是若考虑诸如最大化资源使用率等目标函数为小数的情形时，算法的效率不再有保障。

7. 参考文献

[1] 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.

[2] 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.

[3] Komaki G M, Sheikh S, Malakooti B. Flow shop scheduling problems with assembly operations: a review and new trends[J]. International Journal of Production Research, 2019, 57(10): 2926-2955.

[4] Rossit D A, Tohmé F, Frutos M. The non-permutation flow-shop scheduling problem: a literature review[J]. Omega, 2018, 77: 143-153.

[5] Morrison, D. R., Jacobson, S. H., Sauppe, J. J., Sewell, E. C. (2016). Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. Discrete Optimization, 19, 79-102.

[6] Wolsey, L. A. (2020). Integer programming. John Wiley Sons.

附录 A 问题一的求解代码

1.1 run_this.py

```
from gurobipy import *
import Data
import Block
import ModelBuilder606 as ModelBuilder

if __name__ == '__main__':

    file_name1 = 'attachment1.csv'
    file_name2 = 'attachment2.csv'
    file_name3 = 'attachment3.csv'

    data = Data.Data()
    data.read_data(file_name1=file_name1,
                   file_name2=file_name2,
                   file_name3=file_name3
                   )

    data.identify_control_dependent()
    data.identify_data_dependent()

    # 检查每个基本块的control_dependent_list是否更新
    print('control_dependent_list: ',
        data.block_set[3].control_dependent_list)

    # 检查每个基本块的control_dependent_list是否更新
    print('wr_dependent_list: ',
        data.block_set[4].wr_dependent_list)
    print('ww_dependent_list: ',
        data.block_set[4].ww_dependent_list)
    print('rw_dependent_list: ',
        data.block_set[4].rw_dependent_list)

    # 检查reachable_list
    # print('reachable_list: ', data.block_set[4].reachable_list)
    model_handler = ModelBuilder.ModelBuilder(data=data)

    model_handler.build_model(data=data)
```

```python
import pandas as pd
import Block
from gurobipy import *


class Data(object):
   def __init__(self):
      self.block_set = {}
      self.resource_upper_bound = {'TCAM': 1,'HASH': 2,'ALU':
         56,'QUALIFY':64} # U_r
      self.resource_upper_bound_fold = {'TCAM': 1, 'HASH':3} # Q_r
      self.resource_set = ['TCAM','HASH','ALU','QUALIFY']



   def read_data(self, file_name1='', file_name2='',
      file_name3='', block_num=606):
      file_name1 = file_name1
      file_name2 = file_name2
      file_name3 = file_name3
      # 读取数据1

      attachment1 = pd.read_csv(file_name1, header=0, index_col=0)

      # 读取数据2
      data = []
      ......

      # 读取数据3
      data = []
      f = open(file_name3, 'r')
      lines = f.readlines()
      for line in lines:
         data.append(list(line.strip().split(',')))
      attachment3 = pd.DataFrame(data)

      blockNum = len(attachment1) # 获得基本块的总数
      for i in range(blockNum):
         block = Block.Block() # 每次创新新的block对象

         # 从数据为block添加属性
         block.block_ID = i

         resource_name = ['TCAM','HASH','ALU','QUALIFY']
```

```
        ......

        cnt = 1
        while(cnt < len(attachment3.iloc[i][:])):
            ......

        cnt = 2
        while(cnt < len(attachment2.iloc[2*i][:])):
            if(attachment2.iloc[2*i][cnt] == None):
                break
            block.write_var_list.append(attachment2.iloc[2*i][cnt])
            cnt += 1

        cnt = 2
        while(cnt < len(attachment2.iloc[2*i+1][:])):
            if(attachment2.iloc[2*i+1][cnt] == None):
                break
            block.read_var_list.append(attachment2.iloc[2*i+1][cnt])
            cnt += 1
        self.block_set[i] = block


def identify_control_dependent(self):
    """
    根据依赖关系，计算出控制依赖关系。

    即将data里的 每个block的control_dependent_list修改好。

    同时，顺便更新每个block的reachable_list

    :return:
    """
    # 识别叶子节点
    leaf_set = set()
    blockNum = len(self.block_set)
    for i in range(blockNum):
        if(self.block_set[i].adj == []):
            leaf_set.add(i)

    print('leaf_set: {}'.format(leaf_set))

    """构建网络"""
    Graph = nx.DiGraph()
    # 添加 node
```

```
    ......

    # 添加dummy node，称为1000
    Graph.add_node(1000, x_coor=1000, y_coor=0)
    pos_dict[1000] = (1000, 0)

    # 添加 edge
    for cur in range(blockNum):
        for next in self.block_set[cur].adj:
            Graph.add_edge(cur, next)

    # 连接 leaf node 与 dummy node
    for cur in leaf_set:
        Graph.add_edge(cur, 1000)

    des = 1000
    for org in range(blockNum):
        model = Model('maxFlow')
        model.params.outPutFlag = 0
        x = {}
        y = {}
        FLOW = 10000000

        for edge in Graph.edges():
            x[edge] = model.addVar(lb=0, ub=FLOW,
                vtype=GRB.CONTINUOUS,
                name='x_'+str(edge[0])+'_'+str(edge[1]))
        for node in Graph.nodes():
            y[node] = model.addVar(lb=0, ub=1, vtype=GRB.BINARY,
                name='y_'+str(node))

        ......

        model.optimize()
        # model.write('max_flow.lp')
        print('org: {}, model status: {}'.format(org,
            model.status))

        # 修改data中每个block的control_dependent_list
        ......


def identify_data_dependent(self):
    """
```

```
        根据 write_var_list 和 read_var_list 计算出数据依赖关系。
        即将data里面的 每个block的data_dependent_list修改好。

        :return:
        """
        ......

    def is_writeRead(self, block1, block2):
        for var1 in block1.write_var_list:
            if var1 in block2.read_var_list:
                return True
        return False

    def is_readWrite(self, block1, block2):
        ......

    def is_writeWrite(self, block1, block2):
        ......


class Block(object):
    def __init__(self):
        self.block_ID = 0
        self.consumed_resource = {} # TEAM: HASH:
        self.adj = []              # 邻接节点
        self.write_var_list = []
        self.read_var_list = []
        self.control_dependent_list = [] # 控制依赖
        self.wr_dependent_list = [] # 写后读 数据依赖
        self.ww_dependent_list = [] # 写后写 数据依赖
        self.rw_dependent_list = [] # 读后写 数据依赖
        self.reachable_list = [] # 从该block可达的所有block




from gurobipy import *

class ModelBuilder(object):

    def __init__(self, data):
        self.data = data
        self.model = Model('Chip resource layout')
```

```python
        self.x = {}
        self.y = {}
        self.z = {}
        self.max_stage_num = 607 #len(data.block_set)
        self.big_M = 100000000

    def build_model(self, data=None):
        """
        Build the IP model for the chip resource layout problem.

        :param data:
        :return:
        """

        """ create the decision variables """

        for k in range(self.max_stage_num):
            self.z[k] = self.model.addVar(lb=0, ub=1,
                vtype=GRB.BINARY, name='z_'+str(k))
            self.y[k] = self.model.addVar(lb=0, ub=1,
                vtype=GRB.BINARY, name='y_' + str(k))
            for b in range(len(self.data.block_set)):
                self.x[b, k] = self.model.addVar(lb=0, ub=1,
                    vtype=GRB.BINARY, name='x_' + str(b) + '_' + str(k))


        """ set the objective function """
        print(' -------- Set Objective --------- ')
        obj = LinExpr()
        for k in self.y.keys():
            obj.addTerms(1, self.y[k])
        self.model.setObjective(obj, GRB.MINIMIZE)

        """ add constraints: (1) : y[k] >= y[k+1] """
        print(' -------- Add cons 1 --------- ')
        for k in range(self.max_stage_num - 1):
            self.model.addConstr(self.y[k] >= self.y[k+1],
                name='logic_'+str(k))

        """ add constraints: (2) : sum x[b, k] == 1 """
        print(' -------- Add cons 2 --------- ')
        for b in range(len(self.data.block_set)):
            lhs = LinExpr()
            for k in range(self.max_stage_num):
```

```
        lhs.addTerms(1, self.x[b, k])
    self.model.addConstr(lhs == 1,
        name='allocate_once_'+str(b))

""" add constraints: (3) : sum x[b, k] <= M * y[k]> """
for k in range(self.max_stage_num):
    lhs = LinExpr()
    .......

""" add constraints: (3) : sum d[b, r] x[b, k] <= U[r] """
print(' -------- Add cons 3 --------- ')
for k in range(self.max_stage_num):
    for r in self.data.resource_set:
        .......

""" add constraints: (4) : sum d[b, r] x[b, k] <= U[r] """
print(' -------- Add cons 4 --------- ')
for k in range(16):
    for r in ['TCAM', 'HASH']:
        .......

""" add constraints: (5) : if sum >= 1, then z[k] = 1, k is
    even """
print(' -------- Add cons 5 --------- ')
for k in range(self.max_stage_num):
    .......

""" add constraints: (6) : sum z[k] <= 5 """
print(' -------- Add cons 6 --------- ')
lhs = LinExpr()
for k in range(self.max_stage_num):
    lhs.addTerms(1, self.z[k])
self.model.addConstr(lhs <= 5, name='TEAM_limit')

""" add constraints: (7) : sum k x[i][k] <= sum k x[j][k] -
    1 + M(1 - a[i][j]) """
print(' -------- Add cons 7 --------- ')
for i in range(len(self.data.block_set)):
    print(' block: {} '.format(i))
    for j in range(len(self.data.block_set)):
        if(i != j):
            lhs = LinExpr()
            for k in range(self.max_stage_num):
                lhs.addTerms(k, self.x[i, k])
```

```
            rhs = LinExpr()
            for k in range(self.max_stage_num):
                rhs.addTerms(k, self.x[j, k])

            is_control_dependent = 0
            is_wr_dependent = 0
            is_ww_dependent = 0
            is_rw_dependent = 0
            is_wr_or_ww_dependent = 0
            is_reachable = 0
            ..........

            .......

    """ update the model, avoid lazy update issues """
    self.model.update()


    """ solve the model """
    self.model.setParam('MIPFocus', 1)
    self.model.optimize()

    """ print the results """
    self.print_sol()

def print_sol(self):
    if(self.model.SolCount >= 1):
        print("%20s: %5d" % ('Model status', self.model.status))
        print("%20s: %5d" % ('Objective value',
            self.model.ObjVal))
        print('==================================================')
        print('            芯片资源排布方案              ')
        print('%8s | %-30s' % ('流水线级数', '分配的基本块编号'))
        for k in range(self.max_stage_num):
            print('%12d | ' % k, end='')
            for b in range(len(self.data.block_set)):
                if(self.x[b, k].x > 0.5):
                    print('%4d' % b, end=' ')
            print()
```

附录 B 问题一的求解代码

2.1 run_this.py

```
from gurobipy import *
import Data
import Block
import ModelBuilder606 as ModelBuilder

if __name__ == '__main__':

    file_name1 = 'attachment1.csv'
    file_name2 = 'attachment2.csv'
    file_name3 = 'attachment3.csv'

    data = Data.Data()
    data.read_data(file_name1=file_name1,
                   file_name2=file_name2,
                   file_name3=file_name3
                   )

    data.identify_control_dependent()
    data.identify_data_dependent()

    # 检查每个基本块的control_dependent_list是否更新
    print('control_dependent_list: ',
       data.block_set[3].control_dependent_list)

    # 检查每个基本块的control_dependent_list是否更新
    print('wr_dependent_list: ',
       data.block_set[4].wr_dependent_list)
    print('ww_dependent_list: ',
       data.block_set[4].ww_dependent_list)
    print('rw_dependent_list: ',
       data.block_set[4].rw_dependent_list)

    # 检查reachable_list
    # print('reachable_list: ', data.block_set[4].reachable_list)
    model_handler = ModelBuilder.ModelBuilder(data=data)

    model_handler.build_model(data=data)
```

```python
import pandas as pd
import Block
from gurobipy import *


class Data(object):
   def __init__(self):
      self.block_set = {}
      self.resource_upper_bound = {'TCAM': 1,'HASH': 2,'ALU':
         56,'QUALIFY':64} # U_r
      self.resource_upper_bound_fold = {'TCAM': 1, 'HASH':3} # Q_r
      self.resource_set = ['TCAM','HASH','ALU','QUALIFY']



   def read_data(self, file_name1='', file_name2='',
      file_name3='', block_num=606):
      file_name1 = file_name1
      file_name2 = file_name2
      file_name3 = file_name3
      # 读取数据1

      attachment1 = pd.read_csv(file_name1, header=0, index_col=0)

      # 读取数据2
      data = []
      ......

      # 读取数据3
      data = []
      f = open(file_name3, 'r')
      lines = f.readlines()
      for line in lines:
         data.append(list(line.strip().split(',')))
      attachment3 = pd.DataFrame(data)

      blockNum = len(attachment1) # 获得基本块的总数
      for i in range(blockNum):
         block = Block.Block() # 每次创新新的block对象

         # 从数据为block添加属性
         block.block_ID = i

         resource_name = ['TCAM','HASH','ALU','QUALIFY']
```

```python
        ......

        cnt = 1
        while(cnt < len(attachment3.iloc[i][:])):
            ......

        cnt = 2
        while(cnt < len(attachment2.iloc[2*i][:])):
            if(attachment2.iloc[2*i][cnt] == None):
                break
            block.write_var_list.append(attachment2.iloc[2*i][cnt])
            cnt += 1

        cnt = 2
        while(cnt < len(attachment2.iloc[2*i+1][:])):
            if(attachment2.iloc[2*i+1][cnt] == None):
                break
            block.read_var_list.append(attachment2.iloc[2*i+1][cnt])
            cnt += 1
        self.block_set[i] = block


def identify_control_dependent(self):
    """
    根据依赖关系，计算出控制依赖关系。

    即将data里的 每个block的control_dependent_list修改好。

    同时，顺便更新每个block的reachable_list

    :return:
    """
    # 识别叶子节点
    leaf_set = set()
    blockNum = len(self.block_set)
    for i in range(blockNum):
        if(self.block_set[i].adj == []):
            leaf_set.add(i)

    print('leaf_set: {}'.format(leaf_set))

    """构建网络"""
    Graph = nx.DiGraph()
    # 添加 node
```

```
    ......

    # 添加dummy node，称为1000
    Graph.add_node(1000, x_coor=1000, y_coor=0)
    pos_dict[1000] = (1000, 0)

    # 添加 edge
    for cur in range(blockNum):
        for next in self.block_set[cur].adj:
            Graph.add_edge(cur, next)

    # 连接 leaf node 与 dummy node
    for cur in leaf_set:
        Graph.add_edge(cur, 1000)

    des = 1000
    for org in range(blockNum):
        model = Model('maxFlow')
        model.params.outPutFlag = 0
        x = {}
        y = {}
        FLOW = 10000000

        for edge in Graph.edges():
            x[edge] = model.addVar(lb=0, ub=FLOW,
                vtype=GRB.CONTINUOUS,
                name='x_'+str(edge[0])+'_'+str(edge[1]))
        for node in Graph.nodes():
            y[node] = model.addVar(lb=0, ub=1, vtype=GRB.BINARY,
                name='y_'+str(node))

        ......

        model.optimize()
        # model.write('max_flow.lp')
        print('org: {}, model status: {}'.format(org,
            model.status))

        # 修改data中每个block的control_dependent_list
        ......


def identify_data_dependent(self):
    """
```

```
        根据 write_var_list 和 read_var_list 计算出数据依赖关系。
        即将data里面的 每个block的data_dependent_list修改好。

        :return:
        """
        ......

    def is_writeRead(self, block1, block2):
        for var1 in block1.write_var_list:
            if var1 in block2.read_var_list:
                return True
        return False

    def is_readWrite(self, block1, block2):
        ......

    def is_writeWrite(self, block1, block2):
        ......


class Block(object):
    def __init__(self):
        self.block_ID = 0
        self.consumed_resource = {} # TEAM: HASH:
        self.adj = []              # 邻接节点
        self.write_var_list = []
        self.read_var_list = []
        self.control_dependent_list = [] # 控制依赖
        self.wr_dependent_list = [] # 写后读 数据依赖
        self.ww_dependent_list = [] # 写后写 数据依赖
        self.rw_dependent_list = [] # 读后写 数据依赖
        self.reachable_list = [] # 从该block可达的所有block


from gurobipy import *
import networkx as nx
import copy

class ModelBuilder(object):

    def __init__(self, data):
        self.data = data
        self.model = Model('Chip resource layout')
        self.x = {}
```

```python
        self.y = {}
        self.z = {}
        self.max_stage_num = 607 #len(data.block_set)
        self.big_M = 100000000
        self.solution_file = ''

    def build_model(self, data=None):
        """
        Build the IP model for the chip resource layout problem.

        :param data:
        :return:
        """

        """ create the decision variables """
        for k in range(self.max_stage_num):
            self.z[k] = self.model.addVar(lb=0, ub=1,
                vtype=GRB.BINARY, name='z_'+str(k))
            self.y[k] = self.model.addVar(lb=0, ub=1,
                vtype=GRB.BINARY, name='y_' + str(k))
            for b in range(len(self.data.block_set)):
                self.x[b, k] = self.model.addVar(lb=0, ub=1,
                    vtype=GRB.BINARY, name='x_' + str(b) + '_' + str(k))

        """ set the objective function """
        print(' -------- Set Objective --------- ')
        obj = LinExpr()
        for k in self.y.keys():
            obj.addTerms(1, self.y[k])
        self.model.setObjective(obj, GRB.MINIMIZE)

        lhs = LinExpr()
        #for k in range(self.max_stage_num):
            #lhs.addTerms(1, self.y[k])
        self.model.addConstr(lhs <= self.max_stage_num, name='LB')

        """ add constraints: (1) : y[k] >= y[k+1] """
        print(' -------- Add cons 1 --------- ')
        for k in range(self.max_stage_num - 1):
            self.model.addConstr(self.y[k] >= self.y[k+1],
                name='logic_'+str(k))

        """ add constraints: (2) : sum x[b, k] == 1 """
        print(' -------- Add cons 2 --------- ')
```

```
for b in range(len(self.data.block_set)):
    lhs = LinExpr()
    for k in range(self.max_stage_num):
        lhs.addTerms(1, self.x[b, k])
    self.model.addConstr(lhs == 1,
        name='allocate_once_'+str(b))

""" add constraints: (3) : sum x[b, k] <= M * y[k]> """
for k in range(self.max_stage_num):
    ..........

""" add constraints: (3) : sum d[b, r] x[b, k] <= U[r] """
print(' -------- Add cons 3 --------- ')
..........

""" add constraints: (4) : sum d[b, r] x[b, k] <= U[r] """
print(' -------- Add cons 4 --------- ')
..........
""" 这里换成了另一种的折叠约束 """
for k in range(16):
    for r in ['TCAM']:
        lhs = LinExpr()
        for b in range(len(self.data.block_set)):
            block = self.data.block_set[b]
            lhs.addTerms(block.consumed_resource[r], self.x[b,
                k])
            lhs.addTerms(block.consumed_resource[r], self.x[b,
                k+16])
        self.model.addConstr(lhs <=
            self.data.resource_upper_bound_fold[r],
            name='resource_UB_fold_'+str(k)+'_'+str(r))

""" add constraints: (5) : if sum >= 1, then z[k] = 1, k is
    even """
print(' -------- Add cons 5 --------- ')
for k in range(self.max_stage_num):
    ..........

""" add constraints: (6) : sum z[k] <= 5 """
print(' -------- Add cons 6 --------- ')
lhs = LinExpr()
for k in range(self.max_stage_num):
    lhs.addTerms(1, self.z[k])
self.model.addConstr(lhs <= 5, name='TEAM_limit')
```

```python
""" add constraints: (7) : sum k x[i][k] <= sum k x[j][k] -
    1 + M(1 - a[i][j]) """
print(' --------- Add cons 7 ---------- ')
for i in range(len(self.data.block_set)):
    print(' block: {} '.format(i))
    for j in range(len(self.data.block_set)):
        if(i != j):
            lhs = LinExpr()
            for k in range(self.max_stage_num):
                lhs.addTerms(k, self.x[i, k])

            rhs = LinExpr()
            for k in range(self.max_stage_num):
                rhs.addTerms(k, self.x[j, k])

            is_control_dependent = 0
            is_wr_dependent = 0
            is_ww_dependent = 0
            is_rw_dependent = 0
            is_wr_or_ww_dependent = 0
            is_reachable = 0
            ..........

""" update the model """
self.model.update()

""" 设置log file """
log_file_name = 'P2_607.log'
logfile = open(log_file_name, 'w')
self.model.setParam(GRB.Param.LogFile, log_file_name)


..........

..........

""" set the parameters """
self.model.setParam('MIPFocus', 1)

""" solve the model"""
self.model.optimize()
# self.model.computeIIS()
# self.model.write('model.ilp')
```

```
    print('\n\n======================= Summary
       ===============================')
    ..........

    with open(self.solution_file, 'w') as f: #'x'
       ..........

    """ print the results """
    self.print_sol(solution_file=self.solution_file)

def print_sol(self, solution_file=''):
    if(self.model.SolCount >= 1):
       print("%-32s: %-5d" % ('Model status', self.model.status))
       print("%-32s: %-5d" % ('Objective value',
          self.model.ObjVal))
       print('===============================================================
       print('          芯片资源排布方案(问题2)          ')
       print('%8s | %-30s' % ('流水线级数', '分配的基本块编号'))
       for k in range(self.max_stage_num):
          print('%12d | ' % k, end='')
          for b in range(len(self.data.block_set)):
             if(self.x[b, k].x > 0.5):
                print('%4d' % b, end=' ')
          print()

       # 前面已经打开了，这里只需要append就可以
       with open(solution_file, 'a') as f: #'x'
          f.write("%-32s: %-5d\n" % ('Model status',
             self.model.status))
          f.write("%-32s: %-5d\n" % ('Objective value',
             self.model.ObjVal))
          f.write('==========================================================
          f.write('          芯片资源排布方案(问题2)          \n')
          f.write('%8s | %-30s\n' % ('流水线级数', '分配的基本块编号'))
          for k in range(self.max_stage_num):
             f.write('%12d | ' % k)
             for b in range(len(self.data.block_set)):
                if (self.x[b, k].x > 0.5):
                   f.write('%4d' % b)
             f.write('\n')
```

```python
def groupGeneration(self, lst: object) -> object:
    '''
    input: 一个流水级内部的已经安排好的点
    return: 一个列表，列表中存个各个子列表，一个子列表就是一条同一执行流程

    记得删掉虚拟源点与终点!!!!!!!!!!
    '''
    lst = lst
    grp = []
    Graph = nx.DiGraph()

    # 加 node
    for i in lst:
        Graph.add_node(i, x_coor=i, y_coor=((-1) ** i) * i)

    # 加 edge
    ..........

    # 识别父子节点
    father = []
    child = []
    for node in Graph.nodes():
        if (Graph.in_degree(node) == 0):
            father.append(node)
        elif (Graph.out_degree(node) == 0):
            child.append(node)

    # 添加源 dummy node, 称为1000
    # 添加末 dummy node, 称为1005
    Graph.add_node(1000, x_coor=1000, y_coor=0)
    Graph.add_node(1005, x_coor=0, y_coor=1005)

    # 连接dummy node
    for f in father:
        Graph.add_edge(1000, f)
    for c in child:
        Graph.add_edge(c, 1005)
    GraphCopy = copy.deepcopy(Graph)
    for edge in GraphCopy.edges():
        org = edge[0]
        des = edge[1]
        ..........
```

```
        if (model.SolCount == 0):
            Graph.add_edge(org, des)

    for path in nx.all_simple_paths(Graph, source=1000,
        target=1005):
        path.pop(0)
        path.pop(-1)
        grp.append(path)

    return grp
```