



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

同济大学

参赛队号

22102470072

队员姓名

1. 宋枭炜

2. 陈沛雯

3. 王君豪

中国研究生创新实践系列大赛
中国光谷·“华为杯”第十九届中国研究生
数学建模竞赛

题

目

PISA 架构芯片资源排布优化建模

摘要：

芯片行业由于其具有的技术高密集型和资本高密集性，已然成为衡量各个国家或地区产业综合实力水平的关键产业。PISA(Protocol Independent Switch Architecture) 是当前主流的可编程交换芯片架构之一。在实际 PISA 架构芯片的设计中，为了连线的简化，往往存在着各种资源上的约束，这使得本就有限的芯片资源排布上更为困难。于是，如何设计高资源利用率且低时间复杂度的资源排布算法就对于编译器的设计至关重要。

针对问题一，本文首先对三类附件数据的作用进行分析并进行相应的预处理。针对分析的结果，利用 **0-1 混合整数线性规划模型**，转化问题要求为线性条件，通过**限定不同的流水线最大级数**，求解粗略的解空间，验证相关模型假设，进而引出基于**拓扑排序和贪心策略的启发式搜索模型**。本文利用**拓扑排序、候选基本块回溯祖先树寻找级数下界、平均分流算法快速查询依赖关系、基于二维贪心指标优先队列排序、下界更新和重排**等方法，构建了**依赖约束和资源约束分步优化的启发式模型**。模型在**保证运算速度**的前提下，在问题一中求解得到的最大流水线级数为 42 级，**逼近验证最优解**。同时本文也论证了该启发式模型的**五大合理性和正确性**。

针对问题二，本文将执行流程转化为基本块两点之间可达的问题，原创性地提出了**逆宽度优先搜索动态规划 (RBFS-DP) 算法**和**快速幂 Washall 可达矩阵计算方法**，极大提高了任意两基本块间的可达信息计算速度。同时得力于启发式模型分步优化的**模块性和强结构性**，本文仅通过修改部分资源约束并加入可达计算，就完成了问题二的模型构建。模型在**不损失任何性能**的前提下，求得流水线最大级数为 23 级，**同样逼近验证最优解**。本文最后利用**数学归纳思想**，对问题二模型修改部分进行**严格论证**，证明模型算法策略的必要性和正确性。

关键词： 0-1 整数线性规划 拓扑排序 贪心 平均分流算法 RBFS-DP Washall 可达运算

目录

1 问题重述	4
1.1 问题背景	4
1.2 问题的提出	5
2 模型的假设	6
3 符号说明	6
4 数据分析和预处理	7
5 问题一的求解	8
5.1 问题一分析	8
5.2 基于 0-1 整数线性规划的芯片资源排布方案建模	9
5.2.1 模型建立过程	9
5.2.2 模型结果分析	12
5.3 基于拓扑排序与贪心算法的芯片资源排布建模	13
5.3.1 拓扑排序与回溯	13
5.3.2 判断节点依赖	15
5.3.3 基于贪心算法的最优节点选择与排入	18
5.3.4 满足资源约束并下界更新	21
5.3.5 模型合理性论证	22
5.4 问题一结果与小结	23
6 问题二的求解	26
6.1 问题二分析	26
6.2 基本块可达查询加速算法	26
6.2.1 快速幂 Warshall 可达矩阵加速计算	26
6.2.2 逆向广度优先搜索动态规划（RBFS-DP）加速算法	27
6.3 建立基于执行流程和快速可达计算模型	27
6.4 模型合理性论证	29
6.5 问题二结果与小结	30
7 模型评价	31

7.1 模型的优点	31
7.2 模型的缺点	32
参考文献	33
附录 A MATLAB 源程序	34
A.1 第 1 问程序	34
附录 B Python 源程序	37
B.1 主要函数程序	37

1 问题重述

1.1 问题背景

芯片行业具有技术高度密集性和资本高度密集性，是衡量各个国家或地区产业综合实力水平的关键产业。作为集成电路的载体，芯片目前被广泛用于军工、航空航天、电子产品等各个重要领域，是我国产业结构转型升级的重要支撑 [1]。近年来，人工智能以及物联网技术不断涌现着新的突破，层出不穷的新应用对底层的网络需求也日益多种多样。传统交换芯片往往只具备某项特定的功能，且从研发到使用的时间周期较长。随着软件定义网络的快速发展，网络拥有者在控制面实现了可编程，然而受限于功能固定的传统交换芯片，转发面难以实现可编程的问题成为了制约网络创新与发展的障碍 [2]。

为了充分解放数据平面的编程能力，斯坦福大学的 Nick McKeown 教授团队在 2013 年的 RMT (Reconfigurable Match Tables) 架构基础上研发了一种新的 SDN 数据平面的可编程协议无关交换机架构 PISA (Protocol-Independent Switch Architecture)。随后该团队于 2014 年首次设计并提出了数据平面特定领域编程语言 P4[3] (Programming Protocol-Independent Packet Processors)。在 PISA 架构编程模型中，用户通过编写 P4 程序来定义数据包的处理流程，然后利用 P4 编译器将这段程序翻译成指定网络数据平面的配置信息，从而实现数据面的用户可编程网络数据处理 [4]。在可编程交换芯片内，可编程的解析器和匹配动作单元使得用户能够自定义网络设备的转发行为，芯片的可编程能力极大地扩展了网络处理的灵活性，并且适应各种复杂网络场景的应用，方便用户添加新的网络协议和新的网络功能 [5]。P4 的研究框架如图1.1所示。

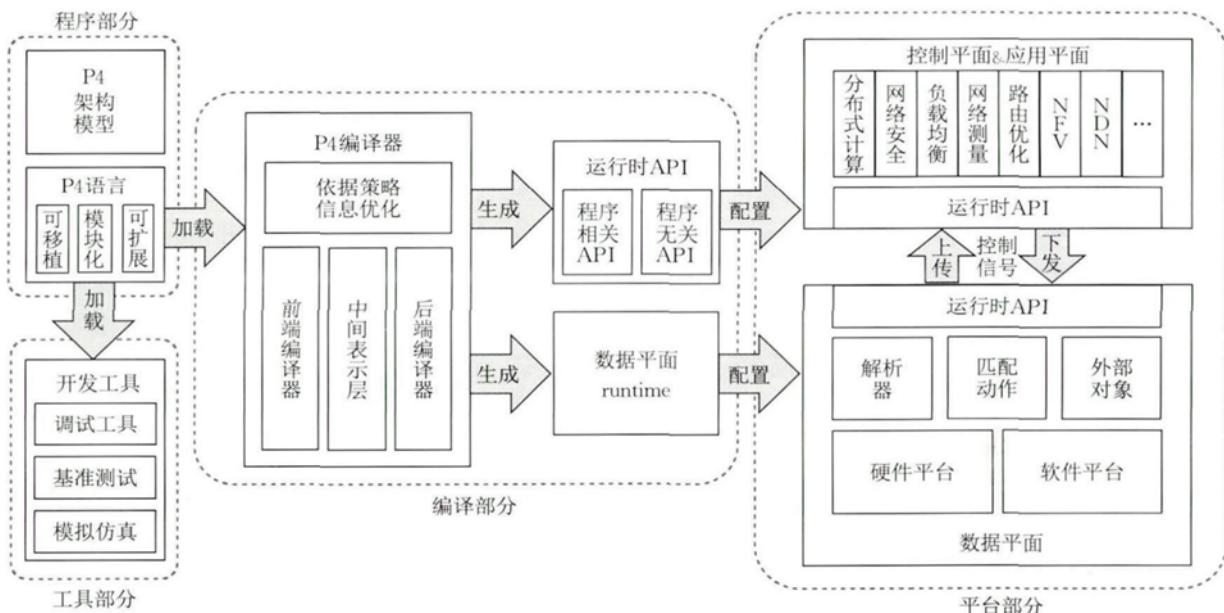


图 1.1 P4 研究框架

如今网络芯片逐渐进入了可编程时代，PISA 已成为当前主流的可编程交换芯片架构之一。PISA 架构如图 2 所示，主要由可编程解释器、可编程的匹配动作单元模块和重组模块构成。到达 PISA 系统的数据包先由可编程解释器解析，再通过入口侧一系列的“匹配-动作”流水线阶段，然后经由队列系统交换，由出口“匹配-动作”流水线阶段再次处理，最后重新组装发送到输出端口 [6]。其中匹配-动作单元模块主要实现各种查找表操作，芯片内部包含多级的匹配动作单元，每一级的匹配动作单元内都包含一定数量的 HASH 资源、SRAM 资源、TCAM 资源和 ALU 资源等。

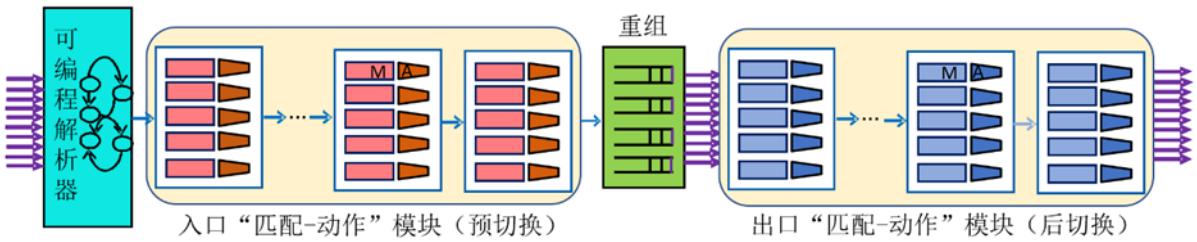


图 1.2 PISA 架构

在对 P4 程序进行编译时，P4 程序会被编译器分成若干个基本块，各基本块按不同的顺序排列在各个层次上。PISA 每个基本块都会占据一定的芯片资源，且为降低连线的复杂程度，常对流水线级内部和级与级之间添加约束条件，确定每个基本块排布到流水线哪一级的 PISA 架构芯片资源排布问题与芯片资源利用率息息相关。此外，由于芯片所需的各类资源都是有限的，并且基本块可能存在数据依赖与控制依赖，因此如何通过排布算法优化排布方案成为提升芯片资源的利用率亟待解决的问题 [7]。

1.2 问题的提出

基于上述研究背景，本文将针对最低级数和最佳空间资源利用率提出优化数学模型，在满足依赖和资源约束的前提下，解决以下两个程序基本块最优资源排布问题：

问题一：题目所给定的 607 个基本块，具有需求资源量、读写变量和指向信息。要求模型通过这些信息得到基本块间的数据依赖和控制依赖关系，并以此为基础确定基本块在流水线级数中的相对大小顺序。同时要求模型在排布时要时刻满足资源约束的要求，在本问题中资源约束可以分为流水线单级资源上限约束、折叠流水线资源上限约束、TCAM 偶数级资源上限约束和基本块排布指派约束。在满足上述依赖和资源约束的前提下，要求模型能够找到占用流水线级数尽量短、流水线空间资源利用率尽量高的基本块排布。同时为了满足高速场景的芯片需求，要求模型所涉及的算法时间复杂度尽量低。

问题二：以问题一的约束和优化目标为基础，问题引入“执行流程”概念，提出不同执行流程的基本块间可以共享部分资源。因此本问题将问题一的硬上限约束转化为了不同执行流程基本块集合所占资源最大值的软上限约束。要求模型在问题一的基础上完成约束

的转换，在保证解的质量的前提下，保证算法的低时间复杂度和高性能表现。

2 模型的假设

1. 所有基本块是单个程序的剖分，仅放入流水线一次；
2. 不考虑某基本块任务执行完成后退出流水线进而发生资源重排的问题，即考虑基本块进入流水线后就不再取出；
3. 单个基本块的所需资源小于任何层级流水线的资源上限；
4. 流水线最高级数越短，流水线空间资源利用率越高，算法时空复杂度越低，模型越好；
5. 基本块间的依赖关系分为在程序执行前即可根据拓扑和读写变量确定的静态数据依赖以及控制依赖，和根据算法执行过程才能确定的动态数据依赖。

3 符号说明

符号	意义
x_{ij}	基本块 i 是否位于流水线 j
y_j	流水线 j 的 TCAM 是否被占用
z_i	基本块 i 位于第几级流水线
T_i	基本块 i 所需 TCAM 资源数
H_i	基本块 i 所需 HASH 资源数
A_i	基本块 i 所需 ALU 资源数
Q_i	基本块 i 所需 QUALIFY 资源数
U	流水线平均资源利用率
m	基本块总数
n	流水线级数
D	基本块的依赖性矩阵
S	各级各基本块 TCAM 资源数构成的 m 行 n 列矩阵
J	各级各基本块 HASH 资源数构成的 m 行 n 列矩阵
B	各级各基本块 ALU 资源数构成的 m 行 n 列矩阵
R	各级各基本块 QUALIFY 资源数构成的 m 行 n 列矩阵

4 数据分析和预处理

本题所给的三种数据分别代表着各基本块所需的资源数量信息、各基本块读写变量信息和各基本块在有向无环图中的邻接基本块信息。

从附件 1 中本文可得到基本块所需的资源数量信息。在已知流水线每级资源上限和折叠级资源上限的前提下，我们可以据此判断基本块位于某级流水线是否满足题目所给定的资源约束条件。在模型中，本文希望能够快速查询某基本块的资源需求，所以需要将原始数据处理为哈希数组，用下标索引来代表基本块，用索引对应的内容表示基本块资源需求，从而实现快速查询。该数据具体结构如图4.1所示：

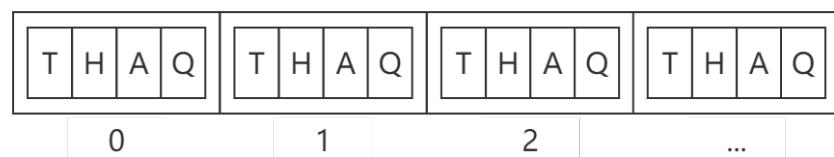


图 4.1 基本块资源需求数据结构表示

从附件 2 中本文可以得到各基本块的读写变量信息。从查看不同基本块间读写变量是否存在差异，可以得到各基本块之间是否存在潜在的数据依赖关系，结合基本块间的拓扑顺序即可确定两基本块是否存在数据依赖和控制依赖。本文结合读写变量和拓扑序得到了任意两个基本块间的依赖分布，其分布矩阵如图4.2所示：

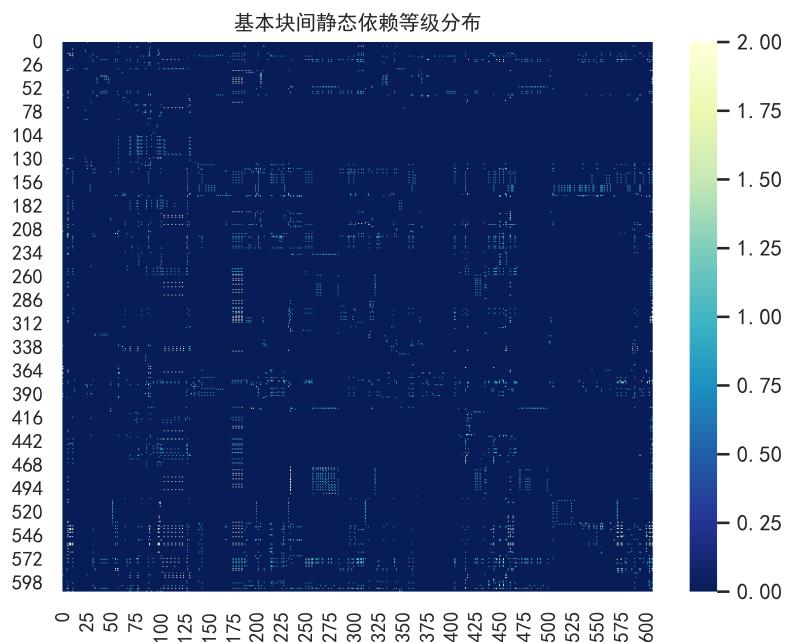


图 4.2 基本块依赖分布矩阵

在图4.2中，颜色为深蓝色的区域代表对应位置两个基本块间不存在依赖关系；颜色为浅蓝色的点代表对应位置的两基本块间存在确定的依赖关系；颜色为白色的点代表对应位置两基本块间存在不确定的依赖关系。可以发现，依赖分布矩阵总体是一个稀疏矩阵，具有依赖关系的区域（包括不确定的依赖）面积仅占分布矩阵总体面积的 1.52%。

从附件 3 中本文可以得到各基本块在流程图中的邻接基本块信息。将基本块抽象为节点，用箭头表示基本块与邻接基本块的跳转方向，即可将该数据转换成一张具有单个起始点的有向无环流程图，如图4.3所示：

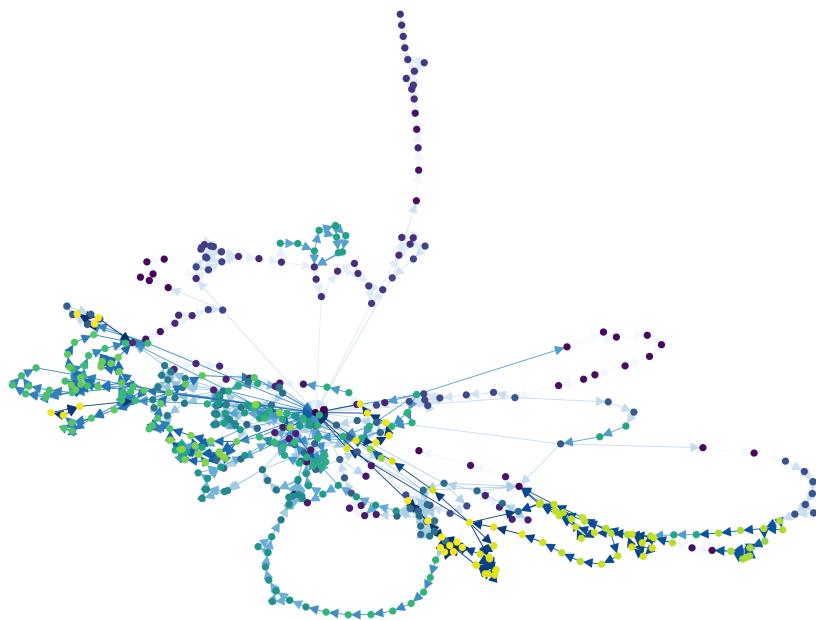


图 4.3 基本块的有向无环流程图

图中共有 607 个节点，969 条跳转有向边。可以从图中的连接分布发现，有些节点具有极多的跳转方向和被跳转方向，呈密集的网状结构；而有些节点只有单一的跳转方向和被跳转方向，呈单链结构。通过 python 的 networkx 库，本文将原始数据转化为了对应的有向图结构，便于后续模型的计算。

5 问题一的求解

5.1 问题一分析

问题一要求在给定的资源约束和依赖约束条件下，以占用的流水线级数尽量短为优化目标，建立合理的资源排布的数学模型，并给出资源排布算法。资源约束包括流水线每级的四个资源上限约束、折叠级资源上限约束、偶数级 TCAM 数量上限约束和基本块指派

单一流水线约束。依赖约束包括基本块间的数据依赖和控制依赖 [8][9]。

如何在不影响芯片工作效率的前提下，资源排布算法尽可能节约资源消耗及流水线的使用是该问题的核心。这促使模型不仅要能得到较优的资源排布，同时模型本身的算法复杂度不能太高，两者缺其一皆会影响芯片的工作性能。

本节首先通过建立 0-1 整数线性规划模型 [10]，通过构建满足问题需求的约束条件，试图找寻芯片资源排布方案的精确最优解。通过线性规划得出的近似下界，本节验证了流水线级数和流水线空间资源利用率的关系假设，并对问题庞大的解参空间得到了进一步的认识。在此基础上，本节建立了基于拓扑排序和贪心算法的启发式算法模型，在有效缩小解参空间的条件下得到了近似的最优结果。本节最后对该模型进行了结果分析和亮点阐述。问题一整体思路如图5.1所示：

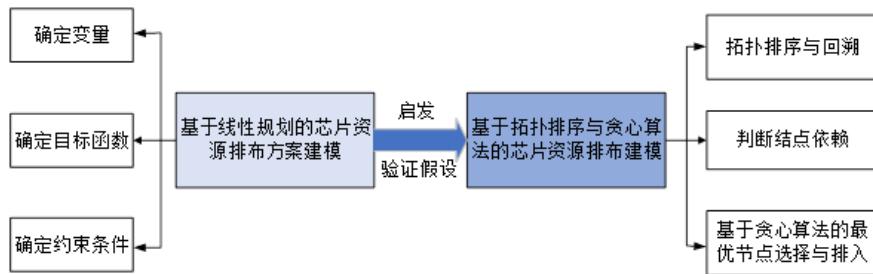


图 5.1 问题一整体思路

5.2 基于 0-1 整数线性规划的芯片资源排布方案建模

5.2.1 模型建立过程

(1) 引入整数变量:

引入 0-1 整数变量 x_{ij} 表示第 i 基本块是否在第 j 级流水线上：

$$x_{ij} = \begin{cases} 1, & \text{第 } i \text{ 基本块在第 } j \text{ 级流水线上} \\ 0, & \text{第 } i \text{ 基本块不在第 } j \text{ 级流水线上} \end{cases}$$

引入 0-1 整数变量 y_j 表示第 j 级流水线上是否有 TCAM 资源：

$$y_j = \begin{cases} 1, & \text{第 } j \text{ 级流水线上有 TCAM 资源} \\ 0, & \text{第 } j \text{ 级流水线上没有 TCAM 资源} \end{cases}$$

引入整数变量 z_i 表示第 i 个基本块在流水线中的级序号。

(2) 建立目标函数:

以流水线平均资源利用率最大为目标函数：

$$\max U = \frac{\sum_{j=0}^n \sum_{i=0}^m x_{ij}(T_i + H_i + A_i + Q_i)}{n(T_M + H_M + A_M + Q_M)} \quad (5.1)$$

其中 m 代表基本块总数，由 attachment1.csv 可知 m 的值为 607。 T_M 、 H_M 、 A_M 与 Q_M 分别代表流水线每级可容纳的最大资源数，由题意可知分别为 1、2、56、64。线性规划的目标为当所有基本块排布到流水线后，所有基本块所用资源占总可用资源的比例最大，即为平均资源利用率最大。由于资源约束不仅仅包括单个的资源上限约束，还有一些相互作用的资源约束，所以 U 的理想值并不能达到 1，但它仍然可以作为线性规划优化的方向。

(3) 创建约束条件：

问题一中并不是所有约束条件都可以直接用线性约束进行表达，故本节需要对部分非线性约束进行线性转化。

- 依赖矩阵约束：

$$D_{ij} = \begin{cases} 0, & \text{第 } i \text{ 基本块与第 } j \text{ 个基本块不存在依赖} \\ 1, & \text{第 } i \text{ 基本块与第 } j \text{ 个基本块存在强依赖} \\ 2, & \text{第 } i \text{ 基本块与第 } j \text{ 个基本块存在弱依赖} \end{cases} \quad (5.2)$$

$$\begin{cases} z_i < z_j, & D_{ij} = 1 \\ z_i \leq z_j, & D_{ij} = 2 \end{cases} \quad (5.3)$$

式中 D_{ij} 代表 m 行 n 列依赖矩阵 \mathbf{D} 的元素，其中 \mathbf{D} 由数据预处理阶段得来。根据题目数据依赖的描述，当第 i 个基本块与第 j 个基本块之间存在读后写依赖或者控制依赖时，第 i 个基本块排入流水线的级数必须小于或等于第 j 个基本块排入流水线的级数，我们将其定义为第 i 个基本块与第 j 个基本块存在弱依赖，在依赖矩阵 \mathbf{D} 中用 2 表示。当第 i 个基本块与第 j 个基本块之间存在写后读或写后写数据依赖时，第 i 个基本块排入流水线的级数必须小于第 j 个基本块排入流水线的级数，我们将其定义为第 i 个基本块与第 j 个基本块存在强依赖，在依赖矩阵 \mathbf{D} 中用 1 表示。

- 基本块指派单一级数约束：

$$\sum_{j=1}^n x_{ij} = 1 \quad (5.4)$$

此约束与题目中问题一的约束 (7) 对应。由于每个基本块只能排布到一级，所以在由 0-1 变量 x_{ij} 组成的矩阵 \mathbf{X} 中，每行元素的和均为 1。

- 四种资源约束：

$$\sum_{i=0}^m S_{ij} \leq 1 \quad (5.5)$$

$$\sum_{i=0}^m J_{ij} \leq 2 \quad (5.6)$$

$$\sum_{i=0}^m B_{ij} \leq 56 \quad (5.7)$$

$$\sum_{i=0}^m R_{ij} \leq 64 \quad (5.8)$$

其中

$$S = T \mathbf{1}_{1 \times n} \odot X \quad (5.9)$$

$$J = H \mathbf{1}_{1 \times n} \odot X \quad (5.10)$$

$$B = A \mathbf{1}_{1 \times n} \odot X \quad (5.11)$$

$$R = Q \mathbf{1}_{1 \times n} \odot X \quad (5.12)$$

式中 T, H, A, Q 均为 m 行 1 列的矩阵，分别表示所有基本块所需的 TCAM、HASH、ALU 及 QUALIFY 资源数。 $\mathbf{1}_{1 \times n}$ 表示 1 行 n 列的全 1 矩阵。与 X 点乘后可分别获得四种资源在每一级内的数量，分别用矩阵 S, J, B, R 表示。以 TCAM 资源为例，将矩阵 S 每列元素相加即可获得流水线每级 TCAM 资源数。同理可得每级中 HASH、ALU、QUALIFY 资源数。此约束与题目中问题一的约束 (1)(2)(3)(4) 对应。分别表示控制流水线每级中 TCAM 资源最大为 1；HASH 资源最大为 2；ALU 资源最大为 56；QUALIFY 资源最大为 64。

- **TCAM 偶数级数量约束：**

$$y_j \mu \leq \sum_{i=0}^m S_{ij} \leq y_j M, \quad i = 0, 1, 2, 3, \dots \quad (5.13)$$

$$\sum_{j=0, j \in Even}^n y_j \leq 5, \quad i = 0, 2, 4, 6, \dots \quad (5.14)$$

式中 0-1 整数变量 y 用于判断第 j 级流水线上是否有 TCAM 资源， μ 是一个充分小的正常数（这里取 10^{-6} ）， M 是一个充分大的正常数（这里取 10^6 ）。当 $y_j = 0$ 时， $\sum_{i=0}^m S_{ij} = 0$ ；当 y_j 为 1 时， $\sum_{i=0}^m S_{ij} > 0$ （大于等于一个极小的正实数，小于等于极大的正实数，等价于大于 0）。将有 TCAM 的且在偶数级的流水线级数量相加小于等于 5 即可满足要求。此约束与题目中问题一的约束 (6) 相对应，即有 TCAM 资源的偶数级数量不超过 5。

- **折叠级数约束：**

$$\sum_{i=0}^m S_{ij} + \sum_{i=0}^m S_{i(j+16)} \leq 1, \quad 0 \leq j < 16 \quad (5.15)$$

$$\sum_{i=0}^m J_{ij} + \sum_{i=0}^m J_{i(j+16)} \leq 3, \quad 0 \leq j < 16 \quad (5.16)$$

此约束与题目中问题一的约束 (5) 对应。由于第 0 级与第 16 级、第 1 级与第 17 级，…，第 15 级与第 31 级分别为折叠级数，要求折叠的两级 TCAM 资源加起来最大为 1，HASH

资源加起来最大为 3。此外，如果需要的流水线级数大于 32 级，则不考虑从第 32 开始的级数的折叠资源限制。

5.2.2 模型结果分析

本节通过 matlab 求解 0-1 整数线性规划模型。该模型是为了试图找寻最优资源排布的精确解，但是由于依赖条件的数量问题（数据依赖约有 1800 个，控制依赖有 588 个）和算力问题，matlab 并不能给出具体的解，只能给出近似的解下界。通过限定不同的最大流水线级数，matlab 给出的对应空间资源利用率下界如图5.2所示：

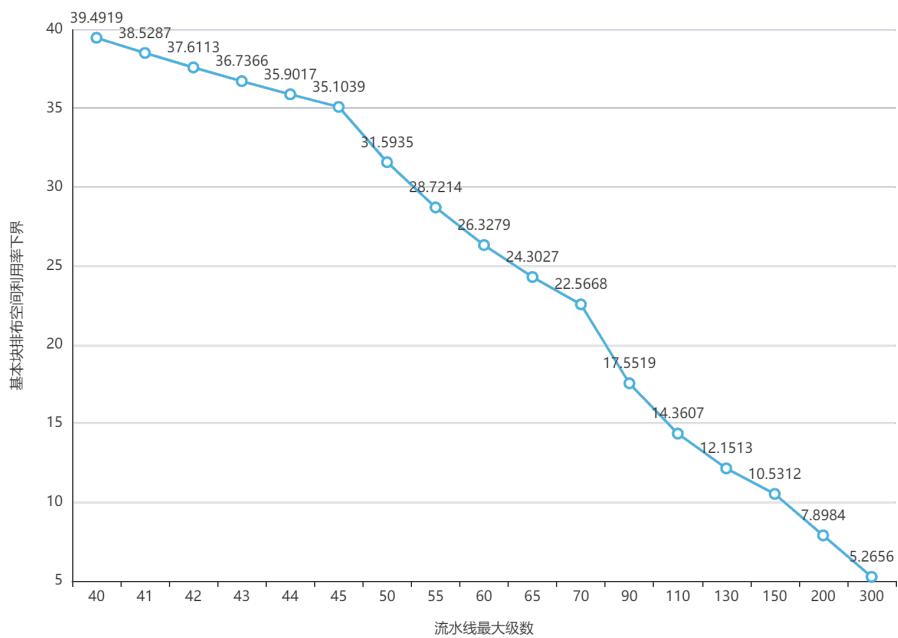


图 5.2 最大流水线级数-空间资源利用利用率对应折线图

可以发现，最大流水线级数与基本块排布空间资源利用率呈明显的负相关关系，验证了流水线级数越低、排布空间资源利用率越高的假设。所以对于问题一而言，以空间资源利用率作为最大化的目标函数，与追求流水线低级数是等价的。

同时为了探究目标函数的权重是否会带来结果的变化，本节也设置了对照实验，对目标函数中的不同资源设置了不同的加权策略，并将结果进行对比。对照实验的动机来源自对同一年级流水线上资源的不同上限的考量，一般认为，资源上限越低的资源更加珍贵，对最终的优化结果带来的影响越大，那么应该在目标函数中赋予它更大的权重。于是本节设置了未加权、对资源上限正向加权和对资源上限负向加权三种目标函数策略作为对照组进行实验，实验结果如图5.3所示。

从实验结果中可以看到，未加权的目标函数（平均资源空间利用率）获得了最高值。因此把未加权的平均资源空间利用率作为优化方向，而不考虑不同资源上限的差异是合理的。



图 5.3 最大流水线级数- 不同加权目标策略对应折线图

的。

同时也可以注意到，由于约束数量的庞大，解决最优排布需要消耗大量的时间。这启发本文开始设计以满足约束和控制空间资源利用率的启发式搜索模型，在最大限度减少参数加快速度的同时，尽可能保证最优排布的质量。

5.3 基于拓扑排序与贪心算法的芯片资源排布建模

根据问题一 PISA 芯片架构排布问题与资源限制，本文提出了一种基于拓扑排序与贪心算法的芯片资源排布建模方法。若本轮拓扑排序层级的所有点视为候选点，则算法总流程包括拓扑排序、候选点祖先树回溯、判断节点依赖、找出基本块允许的级数下界、候选点下界和容量比排序、选择最优候选点、根据资源约束将候选点放入流水线、更新候选点下界并重排 8 个环节。基于拓扑排序与贪心算法的芯片资源排布建模流程如图5.4所示。

5.3.1 拓扑排序与回溯

在一幅有向无环图中，设节点 i 到节点 j 之间存在一条有向边，则称节点 i 是节点 j 的前趋节点，节点 j 是节点 i 的后继节点。对有向无环图进行拓扑排序，是指把图中所有节点排列成一个线性序列，使得对于节点 i 和节点 j ，节点 i 、节点 j 属于该图，并且边 (i, j) 属于该图，则节点 i 在线性序列中出现在节点 j 之前，该线性序列为满足拓扑次序的序列，简称拓扑序列，拓扑排序为由图中节点得到拓扑序列的操作 [11]。若将资源排布过程中的各个基本块抽象为节点，则拓扑排序的算法可描述如下：

- (1) 初始化清空序列；

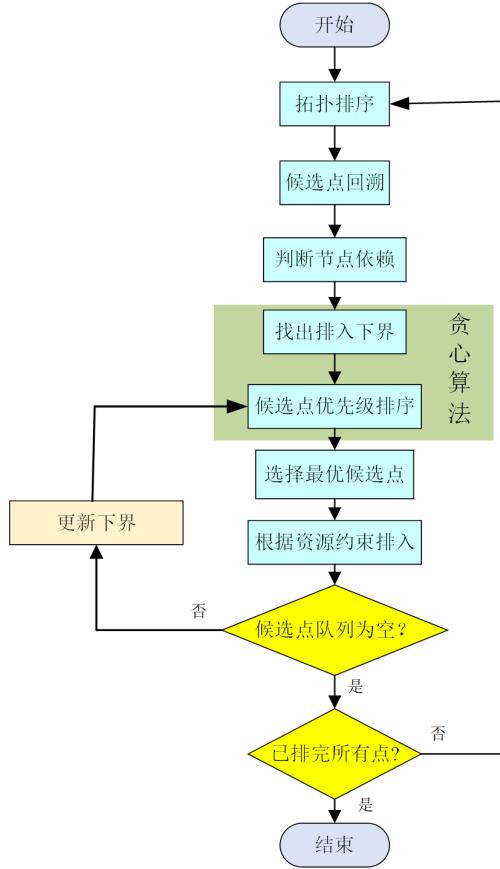


图 5.4 基于拓扑排序和贪心策略模型流程图

- (2) 在图中选取一个无前趋节点的节点进入线性序列队列；
- (3) 删除该节点和以该节点为起点的有向边；
- (4) 重复第 (2)(3) 步，直到图中不存在无前趋节点的节点；
- (5) 得到的线性序列即为拓扑序列。

若使用活动网络 [12](ActivityOnVertexNetwork，简称 AOV-网) 表示拓扑排序，即图中用顶点表示基本块，用有向边来表示基本块的优先关系，每条有向边代表起点对应的基本块是终点对应基本块的祖先节点。由图可直观看出各基本块之间的先后执行流程顺序及跳转关系。例如，如图5.5中所示基本块 C5 的前驱节点为基本块 C2、C1 和 C4，后继节点为基本块 C7 和 C8。AOV 网是不带回路的有向无环图，本文主要通过拓扑排序找出待进行新一轮排入流水线级的基本块。

通过拓扑排序找出新一轮要排入的候选点后，通过逆向宽度优先搜索对所有候选点进行回溯，遍历每个点的所有回溯路径，以找到每个候选点的所有祖先节点，得到一张反向的祖先节点树。

将候选点进行回溯，得到所有祖先的目的在于寻找该候选点所有可能的依赖关系，并求得该点所能放进流水线的级数下界。下面将对候选点反向祖先节点树为何包含所有与该点具有潜在依赖关系的节点进行论证：

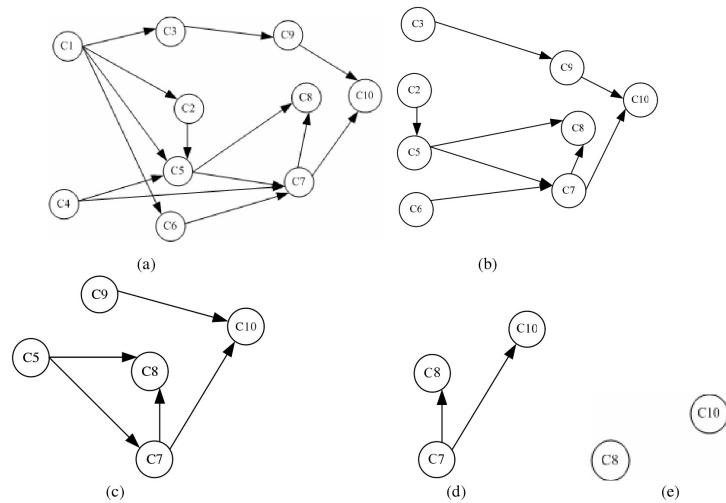


图 5.5 拓扑排序示意图

假设 a 基本块与 b 基本块之间满足依赖，那么就要求 a 基本块要在 b 基本块之前执行，即在程序控制流图中 a 要有一条能够到达 b 的路径，而这便是节点反向祖先树的操作性定义。对于其他拓扑序在该候选基本块之前的并且不是该基本块祖先的节点，他们并不可达该候选基本块，即该基本块的拓扑序并不由它们影响，因此它们不会与该基本块产生依赖关系，如图5.6所示。

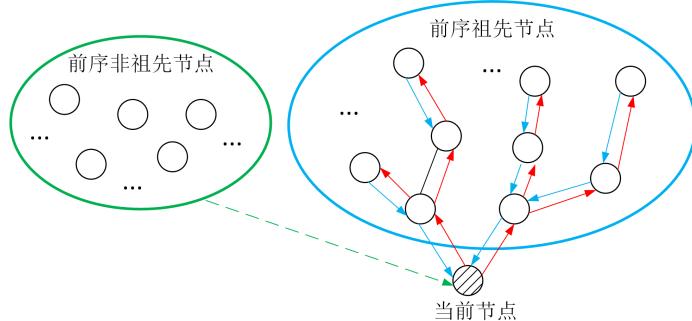


图 5.6 节点回溯寻找前序祖先节点示意图

因此，在节点的祖先树中查找潜在依赖关系即是一种完备策略，不存在其他与该点具有依赖关系的节点被疏漏。于是回溯找到祖先树的方法在保证完备的前提下，减小了需要搜索的节点范围，加快了运算的速度。

5.3.2 判断节点依赖

节点依赖包含数据依赖和控制依赖。其中，数据依赖是由于语句或代码块之间的数据流动而产生的一种约束。它是数据之间的相互关系，反映在一种关系中属性之间的值是否相等。数据依赖是现实世界中属性之间相互关系的抽象，属于数据的固有性质。本文将数据依赖分成静态数据依赖和动态数据依赖。

静态依赖是在已知程序流图的前提下，程序运行前就已经可以确定的依赖关系，它有四种方式：

假定算法基本块为 k_i , k_1 在 k_2 之前执行。

- (1) 当 k_1 写入一个变量, k_2 读取该变量时, k_1 和 k_2 具有写后读的数据依赖关系。
- (2) 当 k_1 读取一个变量, k_2 写入该变量时, k_1 和 k_2 具有读后写数据依赖关系。
- (3) 当 k_1 和 k_2 都写一个变量时, k_1 和 k_2 具有写后写数据依赖关系。
- (4) 当 k_1 只有部分路径经过 k_2 时, k_1 和 k_2 具有控制依赖关系。

动态依赖是本文创新提出的概念。由于芯片流水线实际执行时，流水线级数低的层次会先执行，所以即使两基本块在拓扑序的平级，若两基本块之间存在数据依赖（拓扑序平级的基本块间不可能存在控制依赖），会由于两者放入流水线不同层次的不同情况而发生潜在的依赖关系。由于这种依赖关系要到其中一个基本块最终放入流水线的那个瞬间才能确定，所以本文称其为动态依赖，或者运行时依赖。

为了方便判断依赖，本文算法将以上几个依赖方式总结为 5 类，表示方法为：函数返回值为 0 时，没有依赖关系；1 表示有静态强依赖关系；2 表示有静态弱依赖关系；3 表示动态强依赖关系；4 表示动态弱依赖关系。算法伪代码如下表所示。

算法 1 依赖关系判断算法

输入：基本块号 $k_1; k_2$, 拓扑层次序列 g

输出： k_1, k_2 间是否存在依赖及何种依赖

```
1: Make  $gk_1$  and  $gk_2$  to -1
2: Find  $k_1, k_2$  in which layer of  $g \rightarrow gk_1, gk_2$ 
3: if  $gk_1 > gk_2$  then
4:     return 0
5: else
6:     if  $gk_1 < gk_2$  then
7:         if Find static strong data independence of  $k_1, k_2$  then
8:             return 1
9:         if Find static weak data or control independence of  $k_1, k_2$  then
10:            return 2
11:        if  $gk_1 == gk_2$  then
12:            if Find dynamic strong data independence of  $k_1, k_2$  then
13:                return 3
14:            if Find dynamic weak data or control independence of  $k_1, k_2$  then
15:                return 4
```

控制依赖是程序控制流导致的一种约束。控制依赖定义为：当路径来自一个基本块时，只有部分路径通过下游的基本块，两个基本块构成控制依赖。本文通过一个平均分流算法来判断数据之间是否存在控制依赖。基本思路是，将每个基本块都设置为 0，然后将流图中起点基本块流量赋值为 1，根据基本块之间的联系，将流量平均分给子基本块，再判断子基本块位置来决定是否将均分流量流向它，依次遍历每一个基本块，直到所有基本块都有一个流量值，若两个基本块的流量小于某个阈值，则代表他们之间存在控制依赖。

在单起点的有向无环图（即本问题）中，这是显然成立的；而在多起点的有向无环图中，需要从不同起点出发，将平均分流算法作用于其各自的连通子图，再通过不同连通子图中的重叠点构建整张图的流量分布。关于平均分流算法的流程和示例如图5.7和图5.8所示。

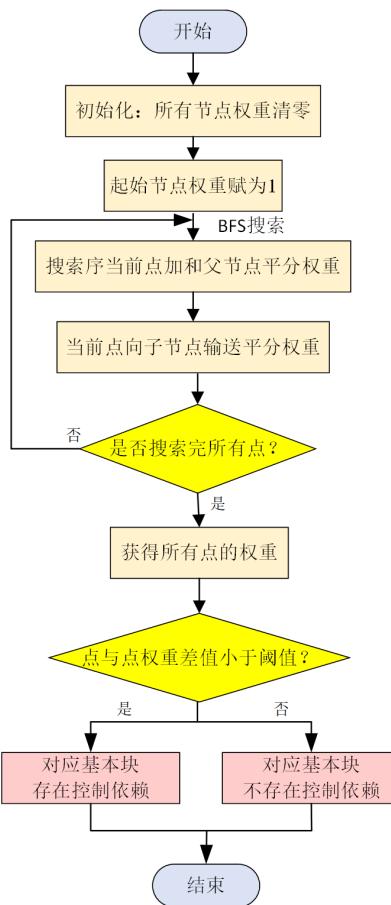


图 5.7 平均分流算法流程图

在算法中，本文规定返回值为 0 时表示没有控制依赖，返回值为 1 时表示存在控制依赖。算法伪代码见算法 2 所示。

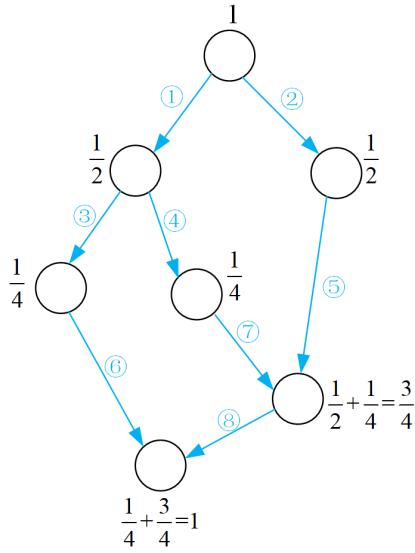


图 5.8 平均分流算法示意图

算法 2 平均分流算法

输入: 基本块 k_1, k_2 , 拓扑层次列表 g , 流图 D

输出: k_1, k_2 之间是否存在控制依赖

```

1: Set the fist node of  $g$  FlowValue 1 in  $D$ 
2: for  $node$  in  $BFS(D)$  do
3:    $FlowValue(node) += Edges[:][node]$ 
4:    $Edges[node][:] = FlowValue(node) / Num(Child(node))$ 
5: if  $FlowValue(k1) - FlowValue(k2) < 10^{-6}$  then
6:   return 1
7: else
8:   return 0

```

5.3.3 基于贪心算法的最优节点选择与排入

贪心算法是指在解决一个问题时，总是做出当前步骤最好的选择。该算法在某种意义上得到的是局部最优解，而不是全局最优解。贪心算法能否得到所有问题的整体最优解，关键是贪心策略的选择。贪心算法一般按如下步骤进行：建立数学模型来描述问题；把求解的问题分成若干个子问题；然后对每个子问题求解，得到子问题的局部最优解；最后把子问题的解局部最优解合成原来解问题的一个解 [13]。

本文借助贪心算法找出排入下界及进行候选点优先级排序，进而实现最优节点的选择与排入。本阶段的贪心策略主要有两个基本准则：一是将基本块放入越低级流水线越好；二是用基本块将流水线塞得越满越好（满足约束的前提下）。

对于第一条准则，由于我们之前已经找到候选基本块和与它满足依赖关系（强依赖与弱依赖）的所有前序祖先节点，那么这就意味着该候选基本块必须满足所有这些依赖关系中最严苛的情况，即这些前序祖先节点在流水线级数的最大值，而该候选基本块必须要大于或者大于等于此最大值，即被我们称为该候选基本块的下界，如图5.9所示。由于该模型是按照拓扑序依次考虑候选点的，所以在考虑某一基本块时，其所有的前序祖先节点必定已经进入流水线，所以下界一定存在。在这里下界就对应将“基本块放入越低级流水线越好”的基本准则。对应所有在优先队列的候选基本块，下界越低的基本块会被优先考虑。

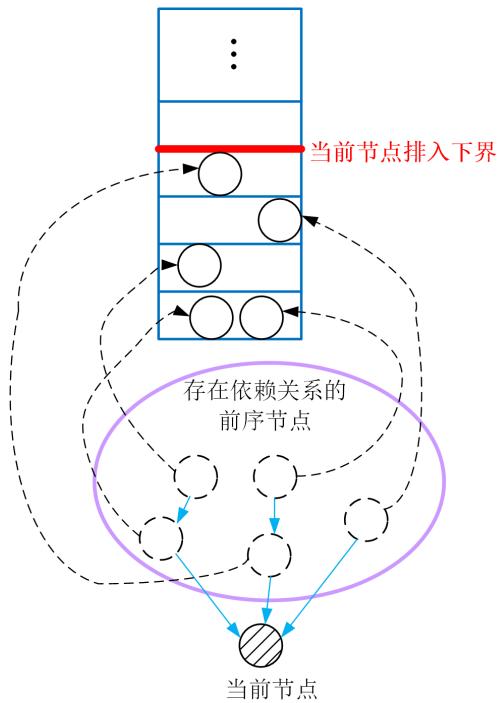


图 5.9 基本块找寻下界示意图

对于第二条准则，本模型选用的指标是基本块所需资源与流水线层级空闲资源的资源比值：

$$E_{ik} = \left| \frac{T_i + H_i + A_i + Q_i}{123 - X_{jk}(T_j + H_j + A_j + Q_j)} - 1 \right|, \quad j = 0, 1, 2, \dots, m \quad (5.17)$$

E_{ik} 代表第 i 个基本块所需资源与第 k 级流水线空闲资源的比值。根据准则二，模型选用的基本块所占用的资源必须最接近但不超过流水线的空间资源，才能尽量让流水线塞满，从图5.10中 3 种策略的比较中可以看出这一点。而对于上述的 E_{ik} 来说，其值就应该尽量大。对于拥有相同下界的候选基本块， E 越大的基本块越优先考虑。

根据上述基本准则，该阶段利用优先队列，在资源排布中的每一步都选择使用最能满足贪心准则的候选点，并试图将其即每一级流水线，在基本块分配中都尽可能地使用现有的资源，增加资源利用率。如图5.11所示。

基于 5.3.2 求出所有依赖关系后，即可判断当前候选点流入流水线的级数。本模型使

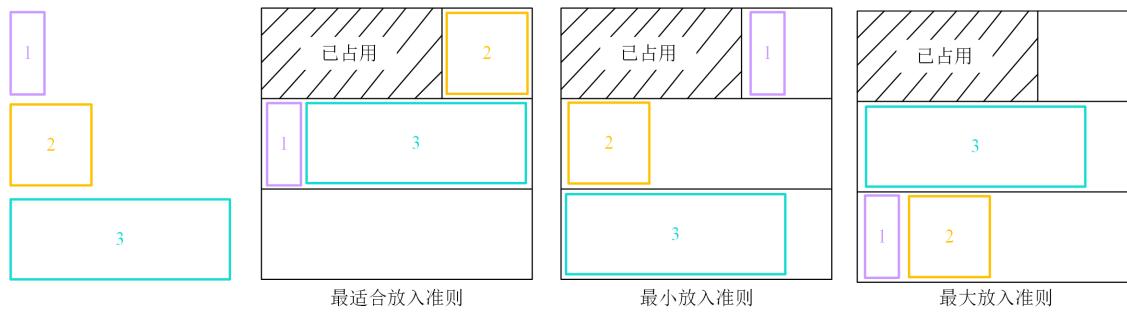


图 5.10 三种放入准则的对比示意图

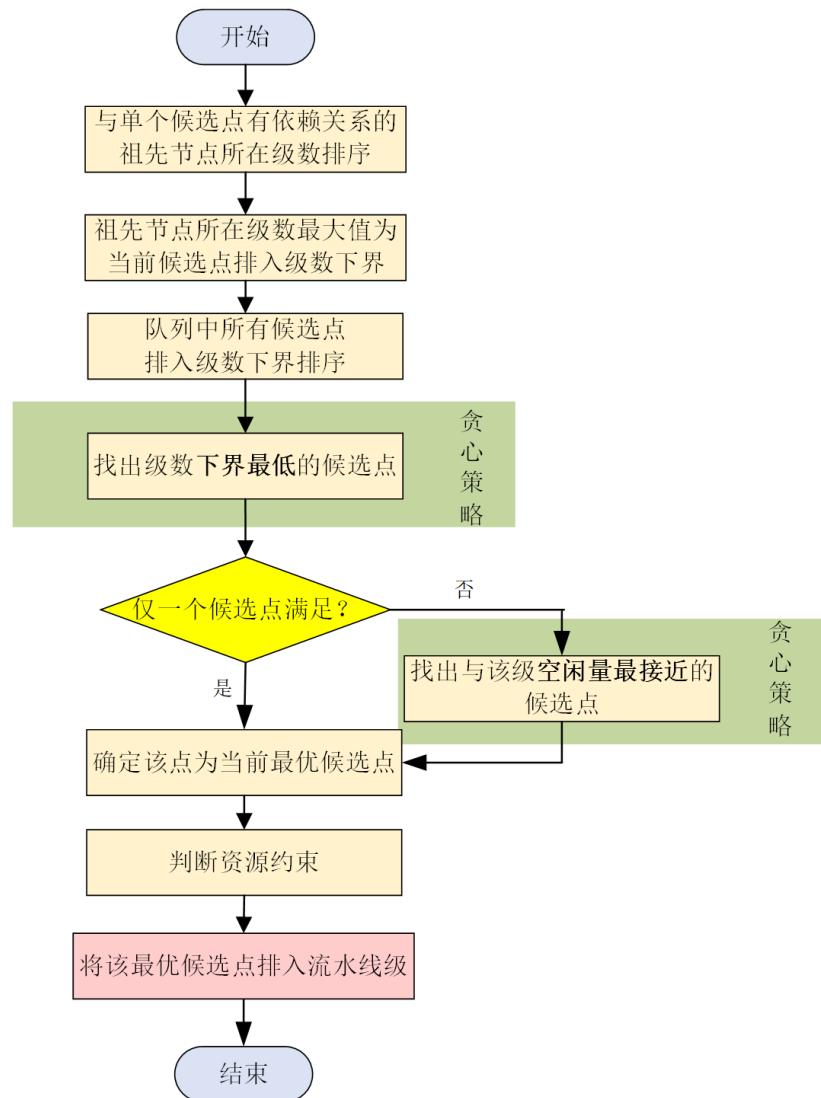


图 5.11 最优点选择排入算法流程图

用贪心策略，以所有与该点有依赖关系的祖先节点所在流水线级的级号的最大值为该点的排入下界，该算法伪代码如下表所示。

算法3 寻找节点排入下界算法

输入：基本块 k ; 层次拓扑列表 g , 流图 D , 基本块放置流水线级号 $Layer$

输出：基本块 k 的下界级号

```
1: Make res1 to [], res2 to []
2: for  $nodeProcessors$  in  $allProcessors$  do
3:   if  $nodeProcessors$  and  $node$  have strong dependence then
4:     insert  $nodeProcessors$  into  $res1$ 
5:   else
6:     insert  $nodeProcessors$  into  $res2$ 
7: Make  $maxA, max1, max2$  to 0
8:  $max1 = max(Layer(res1))$ 
9:  $max2 = max(Layer(res2))$ 
10:  $maxA = max(max1, max2)$ 
11: if  $maxA == max1$  then
12:   return  $maxA + 1$ 
13: else
14:   return  $maxA$ 
```

5.3.4 满足资源约束并下界更新

当候选基本块在优先队列中被选中，即将放入流水线中时，还需考虑资源所带来的约束。当前基本块的下界仅仅是不考虑资源约束时的流水线最低级数，并不代表该级流水线当前允许此基本块的放入。所以在进行基本块放入下界操作前必须先做一次试探检验，判断在资源约束的条件下该级是否允许当前基本块的放入。若允许放入，那么可以直接进行操作；若不允许放入，则需要往更大的流水线级数跳级再进行判断，直到满足允许放入的条件。

当候选基本块成功放入某级流水线后，还需要对位于优先队列的其他基本块进行下界更新和重新排序。因为拓扑排序的同层级基本块之间可能存在动态依赖约束。所以对于已经被放入流水线的当前基本块级号，位于优先队列的其他与该基本块存在动态依赖的基本块皆会转化为静态依赖关系。此时其余基本块就要判断当前基本块所处级号与自身下界的大小关系以及它们之间的依赖关系，并由此更新下界。当所有位于优先队列的基本块更新下界结束后，还需要对这些基本块进行重新排序，进而选择下一轮最新状态的最优基本块。图5.12显示了上述满足资源约束和下界更新的过程。

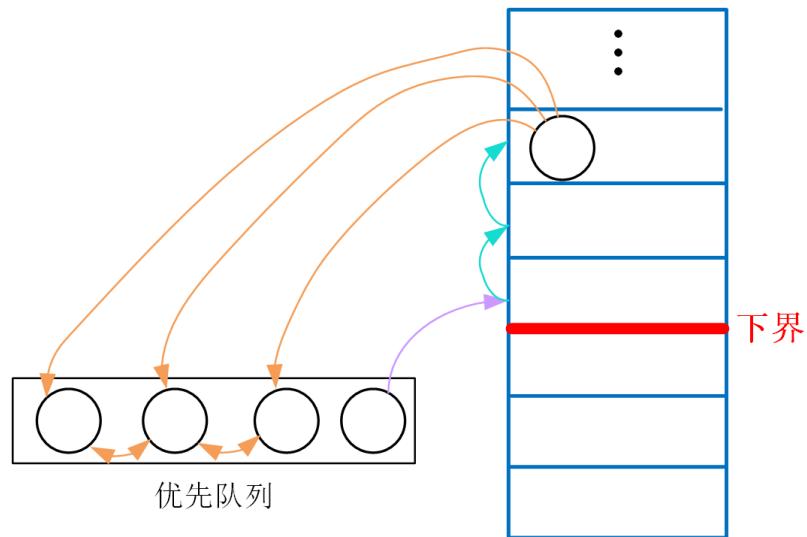


图 5.12 满足资源约束和下界更新重排序示意图

5.3.5 模型合理性论证

本小节将从五方面论证该启发式模型的合理性：

(1) 基于祖先树找寻下界排布策略的合理性

拓扑序自身就满足在流程图中位于较早状态的点可以被较早选择和执行，所以拓扑排序在本问题中显得十分自然。在此基础上，基于回溯查找祖先树的方法可以让模型完备地找到当前基本块的所有具有潜在依赖关系的节点集合。根据这些依赖关系，当前基本块就可以计算出自身的级数下界。并且由于当前基本块的前序节点此时必然已经全部进入流水线，所以下界是确定可以被计算的。同时由于当前基本块必须满足有依赖的前序祖先节点的全部依赖约束，这里包括依赖约束里最严苛的情况，即有依赖的前序祖先节点所在流水线级数的最大值。因此，下界策略的选择是必须且合理的。

(2) 同层级拓扑优先队列准则选择顺序的合理性

同拓扑层次的基本块在经过下界计算后，会进入优先队列进行下界和资源容量匹配度的比较。在该模型中对基本块下界的比较是优先于对资源容量匹配度的比较的。因为如果能让基本块放入更低的流水线级，不仅能够满足问题的要求，还由于低流水线最高级和高资源空间利用率的正相关关系，同时满足了对空间利用率的要求。而资源容量匹配度仅满足了资源空间利用率的局部要求，相较于下界是次优的比较指标。

(3) 将资源约束和依赖约束分阶段求最优的合理性

本问题具有非常多的资源约束和依赖约束，如果通过全局求所有约束共同作用的最优解，则需要非常庞大的算力和较长的时间。但由于拓扑排序的特点，该模型在依次考虑基本块放入流水线的过程中，可以确定地找到所有与之有依赖关系的节点，那么在不考虑资源约束的前提下，该基本块就可以快速地找到自身所能放入流水线级数的最小值，这样就

明显缩小了一部分的搜索空间，加快了搜索的速度。在依赖约束达到最优后，模型再从资源约束的条件下，寻找符合资源约束的最优值，从模型的结果中可以看出这样的操作顺序并不会产生冲突。这样两步寻优的策略显著缩短了计算的时间，同时保证了解的质量。

(4) 区分静态和动态依赖的合理性

在基本块计算下界的过程中，计算得到的皆是静态的依赖约束，因为此时搜索空间内的节点间拓扑顺序，读写信息都是确定的。然而在优先队列排序的过程中，同处于一个优先队列的基本块间由于同层级的拓扑顺序，他们彼此的依赖约束还不明确。但是如果要考虑同拓扑层级间的依赖信息，当它们真正放入流水线时，由于低流水线层级的基本块先运行，会导致原本不明确的依赖约束在此时明确，从而疏漏掉一些基本块间的依赖关系。因此，区分静态依赖和动态依赖，并将其指导基本块放入流水线时的模型算法流程，会使得模型不遗漏任何一个潜在的依赖约束，保证解的正确性。

(5) 模型时间复杂度的合理性

由于该模型应用场景是芯片内部的资源排布，所以对于模型的时间复杂度也有较高的要求。在模型的设计中，本文考虑了许多对模型计算进行加速的策略。如通过基本块回溯找寻祖先节点数缩小搜索范围、依赖约束资源约束分步优化减小解参空间、平均分流算法 $O(E)$ 的流量图构建和后续 $O(1)$ 的超快查询。对时间复杂度的考虑使得模型能以较快的速度得到较优的资源排布。

5.4 问题一结果与小结

表 5.1 问题一实验结果

流水线级数	分配的基本块编号
0	2 3 4 7 10 12 14 15 16 17 18 19 21 22 23 25 27 28 31 33 35 37 38 39 41 43 45 47 49 51 53 54 56 59 61 64 66 69 71 74 75 80 83 88 93 95 96 97 107 111 115 119 123 126 127 130 133 134 135 138 142 146 149 152 155 158 160 162 164 166 167 172 177 180 183 186 187 189 191 193 196 198 206 211 213 219 222 227 230 232 239 241 243 245 247 250 253 256 260 263 265 267 270 273 276 279 282 283 288 291 293 306 309 311 313 314 318 321 323 328 333 338 340 343 346 348 350 353 354 355 358 362 364 365 366 368 369 371 372 373 374 379 380 381 382 384 385 386 387 388 389 390 391 393 394 395 397 399 401 403 407 413 414 420 422 423 426 428 431 432 433 438 440 443 449 452 455 458 460 469 472 475 476 478 480 482 485 488 491 494 497 498 500 504 508 512 515 518 521 525 528 529 534 537 540 541 544 548 549 552 553 557 559 562 566 567 569 570 573 578 581 583 585 588 591 593 599 606

1 20 60 136 137 170 363 377 383 392
2 13 36 139 145 147 163 171 173 303 304 376 378 396 398 400 402 405 545 597
3 48 140 153 165 203 307 335 336 375 406 501 502 509 550 563 568 574 579
4 62 150 161 209 210 240 249 261 285 290 292 299 317 320 329 337 370 408 435
437 448 453 462 463 470 474 490 499 524 531 560
5 72 77 156 159 287 359 409 410 411 412 436 441 442 456 457 468 487 493 596
6 29 73 148 429 434 464 484 496 506 522 532 533 539 546 555 558 577 582 594
595 603
7 154 214 450 451 507 511 530 535 561 564 565 584 586 602
8 1 52 157 424 444 454 459 483 514 519 538 543 600
9 151 415 439 486 492 516 527 536 554
10 30 168 175 297 302 315 331 341 416 445 447 461 467 489 495 523 526 551 556
11 6 129 174 181 339 367 417 421 465 466 473 481 513 517 542 571 576 601
12 8 178 418 446 477 479 510 520 547 572 598
13 9 55 103 108 112 184 212 580 605 607
14 11 57 116 120 185 215 231 326 575
15 34 104 176 188 208 224 294 296 301 330
16
17 201
18 141 310
19 143 144 228
20 190 202 216 218 225 226 298 305 334
21 169 195 200 207 217 221 223 295 300 308 316 324 325 327 357 503
22 197 220 312 332 342 505
23 40 604
24 42 44 233
25 46 50 192 194
26 65 199 229 234 254 257 289 344
27 63 204 286
28 205 251 319 361 471 587
29 32 235 248 252 255 360 425 589 592
30 5 236 237 238 258 259 278 284 404
31 242 272 274 281
32 124 179 182 244 266 269 275 277

33	70 79 81 82 87 105 106 109 114 125 246 262 268 345
34	100 102 110 113 264 271 322 351 356 590
35	76 84 85 117 118 131 280 427
36	67 121 122
37	68 78 128 430
38	86 98 132 347 349 419
39	26 58 94 101 352
40	24 91
41	89 90 92 99

从表中可以看到，该模型得到的流水线最大级数为 42 级。

在求控制依赖的时候时间复杂度是一个特别重要的考虑因素，不管是重建流图还是查询都需要一个低时间复杂度的算法加以支撑。而传统的 DFS 算法的时间复杂度是 $O(N+E)$ 或者 $O(N)$ ，对于稀疏图来说不太友好，而查询所有顶点的复杂度则是 $O(N^2)$ 。我们提出的平均分流算法是通过网络流动态规划的思想，只需要 $O(E)$ 的时间就可以构建整张流图的流量表示。并且通过 $O(N)$ 的空间复杂度的消耗，就可以实现后续 $O(1)$ 的超快查询。对于稀疏流图的超快场景的芯片需求，该算法明显优于传统算法。

另外，基于求祖先节点森林的级数上限方法中，在结合拓扑序的背景下，合理地给定了基本块自身的下限。将超量的依赖约束和资源约束分而治之，使得问题的解参空间从原始的 $O((N+M)^2)$ 变成了 $O(N^2 + M^2)$ ，大大减小了参量的规模，同时也能保证解的质量。由于有向无环图依赖的非传递性和路径性，基于启发式求解策略的祖先森林查询在减少搜索空间的同时，也保证了求解范围的完备性。

而在基于题目问题数据依赖的思考和延伸中，本文创新性地将依赖关系分为静态依赖和动态依赖，区分考量。静态依赖可以在程序加载进芯片时基于流图拓扑序并行计算依赖矩阵，进而在后续实现基本块依赖的 $O(1)$ 查询；本文将动态依赖的计算推迟到基本块排布进流水线的阶段，从而更加精确地描述实际的约束条件。将依赖拆分成上述两个概念区解决，使得问题转化成了 $P + NP$ 问题，保证了解的可行性。

最后，本文发现，同拓扑层次的基本块按照不同顺序进入流水线会带来解的不同。因此对于给定拓扑层次的基本块集合排序算法中，我们自定义了两个指标用于指导基本块的最优排布。一个是前面提到的外接下界，另一个是基本块资源与对应级流水线空闲资源的容量比值。将两种指标共同作用于同层次基本块排序，模型希望基本块最优排布在满足较低级数的要求下，尽可能满足空间资源的最佳利用。

6 问题二的求解

6.1 问题二分析

在问题二中新引进了“执行流程”的概念，如果两个基本块之间有一条或多条执行流程，那么可以等价认为他们是可达的。若两个基本块之间不存在执行流程，那么他们必定通过同一个祖先基本块在选择不同路径的条件下执行，因此他们不可能同时执行，也就为共享资源提供了条件。通过对约束条件的置换，该问题允许不同执行流程上的基本块共享流水线同级和同折叠级资源，因此每级流水线的资源上限不再考虑位于本级流水的全部基本块，而只考虑同执行流程基本块的占用资源最大值，这实际上是对问题一约束条件的放松，所以问题二理应得到比问题一更优的排布。

本问题的关键在于如何快速查询两基本块在流图中的可达性。基于传统搜索的方式会使得查询的时间复杂度近似 $O(n)$ ，不满足高速场景下的芯片需求。因此，本文考虑了基于逆向广度优先搜索动态规划（RBFS-DP）算法和基于快速幂 Warshall 可达计算方法对基本块可达查询进行加速，利用少量的空间消耗和初始化换取了后续 $O(1)$ 的查询速度，使得模型相较于问题一并不会带来显著的性能差距。

由于本文的模型是分阶段启发式模型，对于资源约束的判断发生在选中基本块即将进入流水线排布的阶段。因此通过对该阶段资源约束的更换和可达查询的加入，本文即实现了基于问题二的基本块最优排布模型。

6.2 基本块可达查询加速算法

6.2.1 快速幂 Warshall 可达矩阵加速计算

通过分析第二问中“由一个基本块出发可以到达另一个基本块则两基本块在一条执行流程上，反之不在一条执行流程上”可知，需得到两节点之间是否可以到达这一条件。在解决该问题时，本文借助可达矩阵 [14] 来计算两节点的可达性。可达矩阵不仅是判别一个有向图是否为强连通图或弱连通图的有效工具，并且可以通过自身不断连乘，表示有向图的节点在不同长度路径下所能到达的其他节点信息。利用布尔矩阵的运算性质计算可达矩阵 [15]。

在求解可达矩阵时，Warshall 算法 [16] 可以通过转移矩阵的方式计算出可达矩阵。它的具体步骤是：给定一个行列相等的矩阵，首先找到该矩阵的对角线，并从对角线的左上方开始为第一个元素，然后以对角线上第一个元素为中心，按列展开，寻找中心所在的列中所有不为 0 的元素，接着将“该中心所在的行”加到“该中心所在的列”中所有不为 0 的元素所在的行上。加完之后，以对角线上第二个元素为中心，按列展开，寻找该列中所有不为 0 的元素，然后重复上面“加到所有不为 0 的元素”这一步骤，直到将对角线上所有元素都展开后结束。Warshall 算法通俗的来说就是通过 n 次方操作来判断矩阵中所有可

以联通的点， n 表示节点数量。通过该算法，可以快速得到可达矩阵。为了得到图中任意两节点是否可达的信息，需对 607 行 607 列的可达矩阵进行 n 次幂运算。考虑到芯片资源的实际应用场景，为降低计算的复杂度、提升计算速度，本文使用快速幂（Exponentiation by squaring，平方求幂）算法能够从以往的 $O(N^3)$ 的时间复杂度转化为 $O(N^2 \log N)$ 的时间复杂度计算乘方。

6.2.2 逆向广度优先搜索动态规划（RBFS-DP）加速算法

广度优先搜索 (Breadth First Search, BFS) 是连通图的一种遍历策略，其基本思想是从一个顶点 V_0 开始，辐射状地优先遍历其周围较广的区域。广度优先搜索算法是从初始节点开始，应用产生式规则和控制策略生成第一层节点，同时检查目标节点是否在这些生成的节点中。若没有，再用产生式规则将所有第一层节点逐一拓展，得到第二层节点，并逐一检查第二层节点是否包含目标节点。如此依次层层拓展与检查，直至发现目标节点为止。如果拓展完所有节点，都没有发现目标节点，则问题无解。可以证明，若广度优先搜索能够搜索到解，那么其一定是离搜索起点最短的。

广度优先搜索算法可以从第一层的起始点开始依次向后几层拓展到不同节点。由于问题二的关键在于如何快速查找两个基本块在流图中的可达性，本文结合广度优先搜索算法的特点，使用逆向广度优先搜索 (Reverse Breadth First Search, RBFS) 方法获得基本块的可达信息。逆向广度优先搜索 (Reverse Breadth First Search, RBFS) 方法的基本思路是：由图中的末端节点开始，对每个节点逆向寻找与保存其可达点的信息。末端节点的可达点仅为它本身，而从末端节点开始向上回溯即可获得其父亲节点的可达信息，该父亲节点的可达信息包括了父亲节点本身的可达信息与原末端节点的可达信息，即父亲节点的可达信息比原末端节点的可达信息范围更广。如此依次从末端向始端回溯，再对每个节点收集到的来自其子节点的可达信息进行去重操作，便可以获得每一个节点的可达信息，从而能够判断两个基本块在流图中的可达性。在执行该逆向广度优先搜索算法时，每个节点在向父亲节点传递可达信息的同时也保存了其自身的可达信息，同时父亲节点在获取子节点的可达信息时也不用重新搜索，只需直接查表即可得到答案 (DP)。如此降低了时间复杂度，从而加速了算法执行且符合芯片运作过程中对于快速执行的要求。图6.1展示了执行某个简单有向图的 RBFS-DP 过程。图6.2展示了计算出的可达矩阵热力分布图，其中紫色部分代表基本块 i 与 j 之间可达。

6.3 建立基于执行流程和快速可达计算模型

问题二所更改的是资源相关的约束，而依赖相关的约束并未发生任何更改。并且从流水线层级自身的角度去考察，其对应的资源上限也并未发生任何改变，在本问题中，唯一发生改变且需要考虑的是基本块的行为逻辑，即不同执行流程上的基本块间可以共享资源。所以根据“木桶效应”，影响资源约束条件的决定权不再是流水线，而是转移到了基

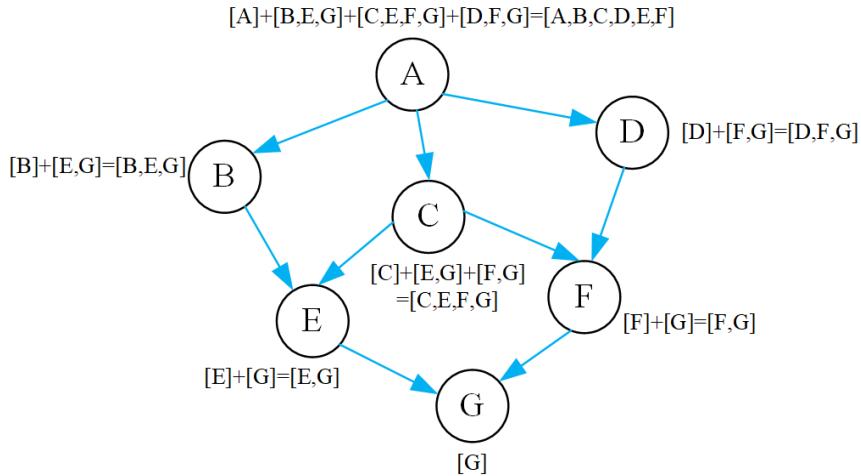


图 6.1 RBFS-DP 示例图

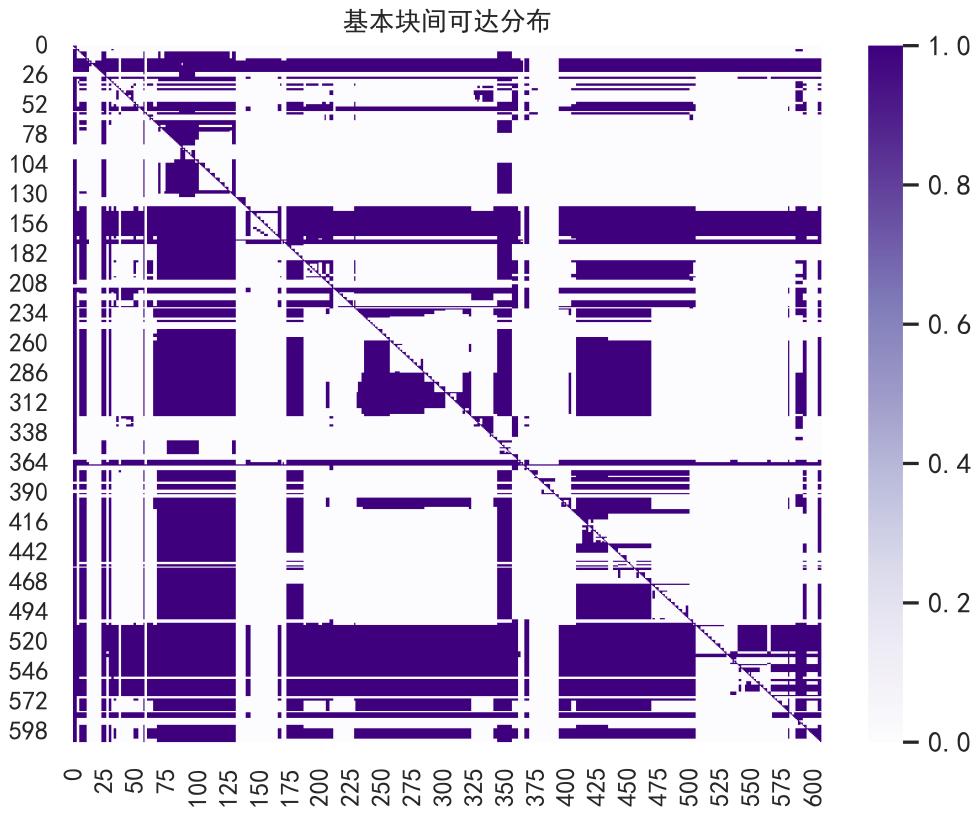


图 6.2 基本块间可达分布热力图

本块执行顺序自身。

本文问题一中的模型是依赖约束、资源约束分开考量，分步优化的模型。结合上述的分析，针对问题二，只需在模型资源约束优化部分修改资源约束条件，并且在得到两个基本块间是否属于同一条执行流程这个问题上加入快速可达计算，模型其他地方不用做任何更改，即可得到问题二的优化资源排布模型。其具体修改内容如下：

- 1) 同拓扑序基本块在优先队列里经过问题一的贪心指标进行排序后, 选择最优的基本块进入流水线。
- 2) 基本块到达其自身级数下界后, 利用快速可达计算, 开始搜索同级流水线中所有与它位于同一条执行流程的基本块集合; 若是位于折叠级, 则开始搜索同级以及对应折叠级上所有与它位于同一条执行流程的基本块集合。
- 3) 计算同执行流程基本块集合在问题二给定的约束资源种类的加和, 以及问题二未特殊说明的其他资源种类在流水线和基本块自身需求资源的加和, 判断其是否超过流水线层级资源上限。若未超过, 代表其满足该层级的资源约束, 可以放入; 若超过, 则代表该层不满足该基本块的放入, 需要去下一层开始重新搜索, 直到找到满足放入条件的流水线层级。

6.4 模型合理性论证

本小节仅对问题二修改的部分进行合理性论证。

本小节需要证明: 基于 6.3 的计算流程, 只可能发生与当前放入基本块同执行流程且位于流水线中的基本块集合特定资源加和会超过流水线层级资源上限。

考虑数学归纳的方法:

- 证明 $n = 1$ 时, 第一个基本块必定能放入。由于第一个基本块放入流水线时是没有任何约束的 (假设基本块需求资源数小于流水线层次资源上限)。那么第一个基本块一定可以放入流水线。
- 假设 $n = k$ 时, 第 k 个基本块已经通过 6.3 的计算流程成功放入流水线。
- 考虑 $n = k + 1$ 时, 第 $k + 1$ 个基本块的放入情况:

当 $k + 1$ 个基本块到达自身级数下界时, 假设该级或折叠级流水线中共有 p 种同一个执行流程的基本块集合 (这些集合相互重叠或包含), 由于当 $n = k$ 时第 k 个基本块已经成功放入, 所以整条流水线处于和谐的状态, 即 p 种集合各自的加和都没有超过流水线层级资源上限。当第 $k + 1$ 个基本块进入流水线后, 只有 q 种基本块集合会发生资源加和的改变, 而这些基本块集合肯定与第 $k + 1$ 个基本块可达, 即它们全体加上第 $k + 1$ 个基本块合并成为了一个全新的集合。而另外 $p - q$ 种基本块集合由于与第 $k + 1$ 个基本块不可达, 所以它们各自的资源加和仍然不变, 保持在 $n = k$ 时的和谐状态。此时流水线中的集合情况可以更改为 $p - q + 1$ 种集合, 由于 $p - q$ 种集合皆不可能发生更改, 所以只可能与第 $k + 1$ 个基本块可达的集合资源加和会超过流水线层级资源的上限。

上述证明验证了基于问题二的模型算法的正确性。基本块进入流水线后, 不需要考虑与它不可达的其他执行流程的资源情况, 只需要把注意力聚焦在与自身可达的那些集合。通过快速可达运算, 获取可达集合 (同一个执行流程的基本块集) 并不会给模型带来计算负担, 在不降低模型性能的同时, 保证了结果的正确性。

6.5 问题二结果与小结

表 6.1 问题二实验结果

流水线级数	分配的基本块编号
0	2 3 4 7 10 12 13 14 15 16 17 18 19 20 21 22 23 25 27 28 30 31 33 35 37 38 39 41 43 45 47 48 49 51 53 54 56 59 61 64 66 69 70 71 72 73 74 80 82 83 88 93 95 96 97 105 106 107 111 113 114 115 119 123 124 125 126 130 133 134 135 136 137 138 139 142 146 147 149 152 155 158 160 162 164 166 167 170 171 172 173 177 180 183 186 187 189 191 193 196 198 203 206 209 211 213 219 222 227 230 232 239 240 241 243 245 247 249 250 253 256 260 263 265 267 270 273 276 279 282 283 288 291 293 306 309 311 313 314 318 321 323 328 335 338 340 343 345 346 347 348 350 353 358 362 363 364 365 366 369 371 372 373 374 379 380 381 382 384 385 386 387 388 389 390 391 393 394 395 396 397 398 399 400 401 402 403 405 406 407 408 410 411 412 413 414 420 422 423 426 428 431 435 436 437 438 440 443 448 449 452 455 458 460 469 471 472 474 475 478 480 481 482 485 486 488 490 491 494 497 498 504 508 509 512 515 518 521 525 528 529 531 534 537 540 541 544 545 548 549 552 553 557 559 562 566 567 569 570 573 578 581 583 585 587 588 591 593 599 606
1	36 60 62 109 110 140 145 153 163 165 210 261 285 303 304 329 333 336 337 349 359 368 370 377 383 392 433 456 468 470 476 477 483 487 493 499 500 501 502 524 550 563 568 574 579 589 592 597 600
2	6 8 75 76 81 100 101 103 104 108 112 117 118 127 128 129 150 161 175 290 292 299 307 317 320 351 352 354 355 356 376 378 415 432 453 457 462 463 479 484 495 496 560 590 596
3	1 9 77 78 79 87 98 116 120 121 122 156 159 181 287 375 409 441 442 450 464 489 594 595 603
4	11 29 92 102 148 178 425 429 434 439 444 447 451 454 459 461 467 492 506 522 532 533 539 546 555 558 577 582 602
5	94 151 184 214 416 424 427 445 466 473 507 511 530 535 561 564 565 584 586
6	52 157 174 417 418 421 430 446 465 514 526 538 543
7	154 419 513 527 536 554 605 607
8	168 185 297 302 315 331 341 516 523 551 556
9	176 339 367 517 519 542 571 576 601
10	84 85 86 131 179 360 510 520 547 572 598

11	24 26 55 58 132 182 212 580
12	57 90 91 99 215 231 326 575
13	34 89 141 188 208 224 289 294 296 301 330
14	143 144 201 207 228 234 286 298 310 319 334
15	190 202 216 218 225 226 235 248 252 255 305
16	217 223 325
17	169 192 194 195 197 200 220 221 295 300 308 316 324 327 332 604
18	32 40 44 50 67 199 229 236 237 242 246 251 254 257 258 259 262 266 272 274 278 281 284 312 357 361 404 503 505
19	5 42 46 65 68 204 233 238 264 269 275 277 322 342
20	63 205 244 268 344
21	271
22	280

从6.1可以得知，问题二优化模型求解的最大流水线级数为23级。问题二约束的放确实带来了更优的空间资源排布方案。

7 模型评价

7.1 模型的优点

针对问题一，本文通过构建0-1整数线性规划模型和基于拓扑排序和贪心策略的启发式模型对问题进行了交叉求解验证。两类模型从两个不同的角度出发：0-1线性规划模型在给定最大流水线级数的条件下搜索解；而启发式模型在构建解的过程中寻求最小的最大流水线级数。通过0-1线性规划的求解，本文获得了大致的解空间的范围，并验证了最大流水线级数与流水线空间资源利用率的负相关关系。这些都为启发式模型的构建奠定了良好的基础。

而本文构建的基于拓扑排序和贪心策略的启发式模型，不仅可以得到近似于全局最优解的资源空间排布方案，还运用了一系列快速运算和剪枝算法，缩小了解参空间；利用依赖约束与资源约束分步优化的启发式模型结构，进一步缩小了模型的检索范围，加快了模型的搜索速度；在计算控制依赖时采用了平均分流算法，参考动态规划和图流的思想，实现了 $O(1)$ 级别的控制依赖查询，进一步加强了模型的计算速度。

同时本文也对模型构建的合理性和正确性进行了详细地论证。通过对拓扑排序本质的研究，本文论证了基于回溯得到祖先树并以此计算基本块级数下界的必要性。同时模型也

创新性地考量了依赖关系的静态动态区分，考虑流水线的实际执行情况，从而更加贴近实际的依赖约束条件。

针对问题二，本文通过将“执行流程”转化为“可达”概念，提出了基于快速可达计算的模型。利用逆宽度优先搜索动态规划(RBFS-DP)算法和快速幂 Warshall 可达矩阵计算方法，实现了对任意两个基本块间是否属于同一个执行流程的 $O(1)$ 查询。

同时由于本文的模型是依赖约束与资源约束分步优化的启发式模型，对于问题二约束的更改，本模型只需要更改资源约束条件计算的部分，而其余地方不需要做任何修改，即可实现问题二的优化排布模型。从中可以看出本模型的通用性和结构性强的特点，同时由于模型是启发式的，也具有较强的可解释性。

最后本文也对问题二模型修改的部分进行了论证，利用数学归纳的方法，证明了模型流程的正确性和必要性，进一步论证了模型的可靠性。

7.2 模型的缺点

模型的拓扑排序只考虑了不同拓扑层次之间的基本块信息，并没有对同一拓扑层次的基本块进行更多的处理。同一层次的基本块间并非是要一起出现并放入优先队列的，由于深度搜索的存在，可能会使得不同拓扑层次的节点同时进入优先队列，而这种方式的模型计算也许会比当前的模型找到更优的解，但本文并未对此进行论证和进一步探究。

参考文献

- [1] 刘琪, 基于 DEA 模型的芯片行业投资策略研究, 浙江大学, 2021.
- [2] 章建钦, 基于可编程交换芯片的 INT 报文过滤模块设计, 电脑编程技巧与维护(6): 19-20, 2019.
- [3] Bosshart P, Daly D, Gibb G, et al., P4: Programming protocol-independent packet processors, ACM SIGCOMM Computer Communication Review, 44(3):87-95, 2014.
- [4] 刘争争, 毕军, 周禹, 等, 基于 P4 的主动网络遥测机制, 通信学报, 39(A01):162-169, 2018.
- [5] 林耘森箫, 毕军, 周禹, 等, 基于 P4 的可编程数据平面研究及其应用, 计算机学报, 42(11):22, 2019.
- [6] 杨泽卫, 重构网络: SDN 架构与实现, 电子工业出版社, 2017.
- [7] 孙康, 可重构计算相关技术研究, 浙江大学, 2007.
- [8] 高念书, 张兆庆, 乔如良, 实用数据依赖分析方法, 计算机学报, 18(4):258-265, 1995.
- [9] 吕蕾, 刘弘, 李鑫, 一种建立控制依赖子图的方法, 计算机工程, 35(15):50-52, 2009.
- [10] Wagner H M, Linear programming techniques for regression analysis, Journal of the American Statistical Association, 54(285):206-212, 1959.
- [11] 王淑芬, 基于拓扑排序的教学计划编制系统的研究与实现, 吉林大学, 2015.
- [12] 朱立华, 王汝传, AOV 网中全拓扑排序算法的设计及应用, 微机发展, 14(12):123-125, 2004.
- [13] Lu C, Liu Q, Zhang B, et al., A Pareto-based hybrid iterated greedy algorithm for energy-efficient scheduling of distributed hybrid flowshop, Expert Systems with Applications: 117555, 2022.
- [14] 杨伟丽, 基于 ISM 有向图的求可达矩阵的简洁算法, 厦门大学, 2007.
- [15] 白冰, 李平, 基于 ISM 的可达矩阵的简便算法, 价值工程, 34(4):213-215, 2015.
- [16] Warshall S, A theorem on boolean matrices, Journal of the ACM (JACM), 9(1):11-12, 1962.

附录 A MATLAB 源程序

A.1 第1问程序

Solution1.m

```
clc,clear;
data = readmatrix('C:\Users\attachment1.csv');
data2 = readmatrix('C:\Users\YilaiMatrix.csv');
data3 = zeros(607, 607);
TCAM = data(:,2);
HASH = data(:,3);
ALU = data(:,4);
QUALIFY = data(:,5);
prob = optimproblem("ObjectiveSense", "max");
flowNum = 90;
miu = 1e-6;
M = 1e6;
RD = repmat(sort(randperm(flowNum)), 607, 1);
x = optimvar('x', 607, flowNum, 'Type', 'integer', 'LowerBound', 0,
    'UpperBound', 1);
y = optimvar('y', 1, flowNum, 'Type', 'integer', 'LowerBound', 0, 'UpperBound', 1);
z = optimvar('z', 607, 1, 'Type', 'integer', 'LowerBound', 1, 'UpperBound', flowNum);
con1 = [];
con2 = [];
for j = 1:607
    for i = j:607
        if(data2(i,j)==1) % i要在j前面
            con1 = [con1; z(i,:)<=z(j,:)-1];
        elseif(data2(i,j)==2) % i要在j前面或者相同位置
            con2 = [con2; z(i,:)<=z(j,:)];
        end
    end
end

con2
con1;
con3 = [];
for i = 1:607
```

```

con3 = [con3; sum(x(i,:).*RD(i,:))==z(i,:)];
end

con4 = [sum(x, 2) == 1];
con5 = [
    sum(x.*repmat(TCAM, 1, flowNum)) <= 1;
    sum(x.*repmat(HASH, 1, flowNum)) <= 2;
    sum(x.*repmat(ALU, 1, flowNum)) <= 56;
    sum(x.*repmat(QUALIFY, 1, flowNum)) <= 64;

];

con6 = [
    miu*y <= sum(x.*repmat(TCAM, 1, flowNum));
    sum(x.*repmat(TCAM, 1, flowNum)) <= M*y;
];
con7 = sum(y(1:2:flowNum)) <= 5;
con8 = [];
if(flowNum>=17 && flowNum<=32)
    for idx = flowNum:-1:17
        con8 = [con8; sum(x(:,idx).*TCAM)+sum(x(:,idx-16).*TCAM)
            <=1];
        con8 = [con8; sum(x(:,idx).*HASH)+sum(x(:,idx-16).*HASH)
            <=3];
    end
elseif(flowNum>32)
    for idx = 32:-1:17
        con8 = [con8; sum(x(:,idx).*TCAM)+sum(x(:,idx-16).*TCAM)
            <=1];
        con8 = [con8; sum(x(:,idx).*HASH)+sum(x(:,idx-16).*HASH)
            <=3];
    end
end

prob.Objective = sum(x.*(repmat(TCAM+HASH+ALU+QUALIFY, 1, flowNum))
    , "all") / (flowNum*(1+2+56+64));
prob.Constraints.con1 = con1; % 写后读/写后写/控制依赖约束
prob.Constraints.con2 = con2; % 读后写依赖约束
prob.Constraints.con3 = con3; % x与z绑定
prob.Constraints.con4 = con4; % 指派约束

```

```
prob.Constraints.con5 = con5; % 资源约束
prob.Constraints.con6 = con6; % 额外约束1 (偶数级有TACM的数目不大于
    5)
prob.Constraints.con7 = con7; % 额外约束1
if(flowNum>=17)
    prob.Constraints.con8 = con8; % 额外约束2 (折叠级约束)
end

sol = prob.solve();
sol.z
```

附录 B Python 源程序

B.1 主要函数程序

Solution.py

```
# 判断两个基本块之间是否有数据依赖 (k1->k2 k1要在k2之前运行)
# 数据依赖分为静态数据依赖和动态运行时数据依赖，静态数据依赖是在拓扑排序时就已经给出，
# 而动态运行时数据依赖是要到最后放入流水线那一步才能知道确定的
# 依赖关系
# 返回值：0：没有依赖关系 1：有强依赖关系 2：有弱依赖关系 3：
# 动态强依赖关系 4：动态弱依赖关系
def isDataDependent(k1: int, k2: int, generations: list):
    if(k1 == k2): return 0
    k1_g = -1
    k2_g = -1
    for idx, g in enumerate(generations):
        if(k1 in g):
            k1_g = idx
        if(k2 in g):
            k2_g = idx
        if(k1_g!=-1 and k2_g!=-1): break
    if(k1_g > k2_g): return 0 # k1的拓扑级别比k2高，不可能存在
    # 依赖关系
    else:
        if(k1_g < k2_g): # k1的拓扑级别比k2低，查找有无写后
            # 读，写后写，读后写数据依赖
            return isDataDependentCore(k1, k2, 0)
        if(k1_g == k2_g): # k1拓扑级别与k2相等，查找潜在的数据
            # 依赖关系
            return isDataDependentCore(k1, k2, 1)

# 判断数据依赖核心代码（有无写后读，写后写，读后写数据依赖关
# 系） k1->k2
# 返回值 0：没有依赖关系 1：有强依赖关系 2：有弱依赖关系
def isDataDependentCore(k1: int, k2: int, isequal: bool):
    k1_R = []
    k1_W = []
    k1_complete = False
```

```

k2_R = []
k2_W = []
k2_complete = False
for idx, item in enumerate(R_W):
    if(int(item['name'])==k1):
        k1_R = item['R']
        k1_W = item['W']
        k1_complete = True
    if(int(item['name'])==k2):
        k2_R = item['R']
        k2_W = item['W']
        k2_complete = True
    if(k1_complete and k2_complete): break
# 写后读
if(len(list(set(k1_W) & set(k2_R)))!=0): return 3 if
    isequal else 1
# 写后写
if(len(list(set(k1_W) & set(k2_W)))!=0): return 3 if
    isequal else 1
# 读后写
if(len(list(set(k1_R) & set(k2_W)))!=0): return 4 if
    isequal else 2
# 没有依赖关系
return 0

# 平均分流算法判断数据之间是否存在控制依赖
# 返回值 0: 没有控制依赖 1: 有控制依赖
def isControlDependent(k1: int, k2: int, DiGraph: nx.DiGraph,
generations: list):
    if(k1==k2): return 0
    flow_list = [0 for i in range(0, 607)]
    flow_list[generations[0][0]] = 1
    edge_list = [e for e in nx.bfs_edges(DiGraph, generations
[0][0])]
    toList = [edge_list[0][1]]
    toNum = 1
    Last = edge_list[0][0]
    for edge in edge_list[1:]:
        if(edge[0]!=Last):

```

```

        for toNode in toList:
            flow_list[toNode] += (flow_list[Last] / toNum)
            toList = []
            toNum = 1
            Last = edge[0]
            toList.append(edge[1])
    else:
        toList.append(edge[1])
        toNum += 1
for toNode in toList:
    flow_list[toNode] += (flow_list[Last] / toNum)
if(abs(k1 - k2)<1e-9): return 1
else: return 0

# 判断是否能将第Node个基本块放入第k级流水线
# 能就放入流水线，更新流水线状态和点分配状态，返回1
# 不能就不执行任何操作，返回0
def ComparePutIn(Node: int, k: int, isOperator: bool, forced: bool):
    global num_TCAM_onEven
    global flow_max
    if(k > flow_max): return 0
    Node_source = Source[Node]
    if(ComparePutInCore(flow_Source[k], Node_source, k)==1): #
        经判断可以放入
        if(isOperator):
            # 执行放入操作
            flow_Source[k] = [i + j for i, j in zip(
                flow_Source[k], Node_source)]
            Node_allocated[Node] = 1
            Node_whichFlow[Node] = k
            if(k%2==0 and Node_source[0]>0): num_TCAM_onEven
                += 1
    return 1
elif(forced):
    while(ComparePutInCore(flow_Source[k], Node_source, k)
        ==0):
        k += 1
        # print(k, Node_source)

```

```

        if(flow_max < k):
            flow_max = k
    # 执行放入操作
    flow_Source[k] = [i + j for i, j in zip(flow_Source[k],
                                              Node_source)]
    Node_allocated[Node] = 1
    Node_whichFlow[Node] = k
    if(k%2==0 and Node_source[0]>0): num_TCAM_onEven += 1
    return 1
else:
    return 0

# 判断是否能将第Node个基本块放入第k级流水线 核心代码
# 资源约束 + 折叠资源约束 + TCAM偶数级约束
# 返回值: 0: 不可以 1: 可以
def ComparePutInCore(flowSource: list, NodeSource:list, k: int):
    global num_TCAM_onEven
    assert len(Source_Upper) == len(flowSource)
    assert len(Source_Upper) == len(NodeSource)
    flag = True
    res = [i + j for i, j in zip(flowSource, NodeSource)]
    for idx,_ in enumerate(Source_Upper):
        if(Source_Upper[idx]<res[idx]):
            flag = False
            break
    if(k>=16 and k<=31):
        res = [i + j + k for i, j, k in zip(flowSource,
                                              flow_Source[k-16], NodeSource)]
        for idx,_ in enumerate(Source_UpperZhедie):
            if(Source_UpperZhедie[idx]<res[idx]):
                flag = False
                break
    if(k%2==0 and NodeSource[0]>0 and num_TCAM_onEven>5): flag
        = False
    return flag

# 考虑静态依赖 闭区间(可取到的)
def findNodeLowBound(node: int, DiGraph: nx.DiGraph,

```

```

generations: list):
    global flow_max
    res1 = [] # 具有强依赖的
    res2 = [] # 具有弱依赖的
    allProcessors = findAllPredecessors(node, [e for e in nx.
        bfs_predecessors(DiGraph, generations[0][0])])
    # allProcessors = findAllNode(node, generations)
    for nodeProcessors in allProcessors:
        if(isDataDependent(nodeProcessors, node, generations)
        ==1):
            res1.append(nodeProcessors)
        elif(isDataDependent(nodeProcessors, node, generations)
        ==2 or isControlDependent(nodeProcessors, node,
        DiGraph, generations)==1):
            res2.append(nodeProcessors)

max1 = 0
max2 = 0
for node in res1:
    max1 = max1 if max1 > Node_whichFlow[node] else
        Node_whichFlow[node]
for node in res2:
    max2 = max2 if max2 >= Node_whichFlow[node] else
        Node_whichFlow[node]

max1_increased = False
while(ComparePutIn(node, max1, 0, 0)==0 if max1==0 else
    ComparePutIn(node, max1+1, 0, 0)==0):
    max1 += 1
    max1_increased = True
    if(max1 > flow_max):
        flow_max = max1 + 1
while(ComparePutIn(node, max2, 0, 0)==0):
    max2 += 1
    if(max2 > flow_max):
        flow_max = max2
max1 = max1 + 1 if max1_increased == 0 and max1 != 0 else
    max1
return max1 if max1 >= max2 else max2

# 利用逆向BFS_DP构建可达表

```

```

def KedaDP(DiGraph: nx.DiGraph, generations: list):
    res = [[i] for i in range(0, 607)]
    edge = [e for e in nx.bfs_predecessors(DiGraph,
                                             generations[0][0])]
    edge.reverse()
    for e in edge:
        res[e[1]] = res[e[1]] + res[e[0]]
    # print(res)
    return res

# 判断两点之间是否在同一条路径上（是否可达）
# 返回值 0: 不在 1: 在
def KeDa(k1: int, k2: int, DiGraph: nx.DiGraph, generations:
         list):
    kedaBiao = KedaDP(DiGraph, generations)
    return k2 in kedaBiao[k1] or k1 in kedaBiao[k2]

# 快速幂warshall计算可达矩阵并建立拓扑序
def Quick_Tuopu(Edges: list):
    nodenum = 607
    m = np.matrix(np.zeros((nodenum, nodenum)), dtype = bool)
    for e in Edges:
        m[e[0], e[1]] = 1
    I = np.matrix(np.identity(len(m)), dtype = bool)
    m = m + I
    res = m
    while(nodenum - 1):
        if(nodenum & 1): res = res * m
        m = m * m
        nodenum = nodenum >> 1
    # print(res[0,:])
    # np.savetxt(fname="KeDaMatrix.csv", X=np.array(res), fmt
    # ="%d", delimiter=", ")
    return res

def Quick_Keda(k1: int, k2: int, Edges: list):
    KedaBiao = Quick_Tuopu(Edges)
    return KedaBiao[k1, k2] or KedaBiao[k2, k1]

```