南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

算法导论大作业报告

我们应该怎么进行小组作业——最大流问题基础与扩展

陈兴浩

年级：2023 级

专业：计算机科学与技术

指导教师：苏明

2025 年 6 月 10 日

摘要

本文针对大学生小组作业分配这一现实问题，采用网络流理论构建了一套系统化的解决方案。文章通过层层递进的方式，分别解决了三个不同复杂度的任务分配问题：基础的 DDL 约束下任务分配、考虑时序约束的任务调度，以及融入组员情绪因素的现实分配问题。

在算法实现方面，本文详细介绍了 SPFA 算法、最小费用最大流算法和 Dinic 算法的原理与应用。针对第一个问题，采用最小费用最大流模型，将每个组员拆分成多个节点表示不同的任务执行顺序，实现了平均完成时间的最小化。对于第二个问题，引入了差分速度分层技术解决时间离散化建模中的资源冲突问题，通过二分搜索结合最大流算法求解最小延迟时间。最后针对现实情况，建模了组员的分段线性愤怒值函数，通过网络流的拆边技术精确表达了非线性目标函数，实现了组员满意度的整体最优化。

实验结果表明，所提出的算法能够在合理的时间复杂度内求解各类任务分配问题，为小组作业的科学分工提供了理论基础和实用工具。本研究将网络流理论与实际应用相结合，展现了算法在解决复杂优化问题中的强大能力。本报告有关代码和测试数据均会上传至**Gitlab 链接**。

关键字：网络流；任务分配；最小费用最大流；SPFA；时序调度；分段函数建模

目录

一、 实验环境

本次实验运行在 x86 AMD CPU 平台上，使用 g++ 编译器对所用代码进行编译，从而保证客观地分析代码实际的性能和正确性。所用平台基本信息如下所示。

CPU 型号	AMD Ryzen 7 4700U with Radeon Graphics
CPU 主频	2.00GHz
编译器	g++.exe (x86_64-win32-seh-rev0, Built by MinGW-W64 project) 8.1.0

表 1: x86 平台实验环境展示

二、 问题描述

(一) 问题背景

在进入大学之后，每一个南开大学的大学生都不得不面对如何解决小组作业的问题！小组作业的一个必然要求即为合理地、根据不同合作伙伴的特点和性格分配合理科学的工作，但事实上很多大学生都不具备这一能力，导致作业速度大大降低，而且影响同学之间的友好关系（哭）。

为此，我们希望构建一套数学化、系统化的分工算法，为正在为小组作业头疼的同学们提供科学的指导。鉴于这一问题既涉及任务次序的安排，也涉及多人人力资源的协同，可借助网络流等算法模型来加以解决。接下来，我们将通过层层深入的讨论，使用本学期算法导论讲解的内容，逐步搭建并完善这一解决方案。

(二) 问题描述

我们采用步步深入的方法，由简到繁地解决这一问题，因此，我们将逐渐给问题加入真实的情境，首先从最简单的问题出发，最终得到切实可行的计划。

但由于算法的特性，我们必然需要首先知道一系列前置信息，这需要结合实际情况去搜寻，但我们保证，所需的变量均为容易收集的值。

首先考虑最简单最抽象的小组的任务分配问题：如何能在 ddl 前完成任务？显然小组作业应对的是一个大问题，我们需要将问题拆分成多个子问题，随后分配给多个同学。在此问题中，我们不考虑任何的其他因素，只将影响因素抽象为完成对应任务的时间，不考虑任何其他因素，解决最容易的问题。

我们进一步复杂这个问题，可以发现事实上任务不可能是这样并行完成的，肯定是有先后的完成顺序，此时我们就不得不将子任务的相互关联关系纳入考虑。可以总结出以下规律：某些子任务只能在项目推进到某个阶段之后才能着手（即它们有各自最早开始时间）；每个子任务都有固定的工作量，必须要投入等量的有效工作时间才能完成；课程往往设有硬性的 DDL（截止时间）——子任务也带着各自的最晚完成时刻。

再加上现实中同学数量有限、同一时刻一个人只能专注于一件事。问题就落在：给定这些最早开始—最晚结束—工作量三元组，以及固定的人手数，究竟能否排出一张“谁在什么时间段做什么”的时间表，使所有子任务都在各自 DDL 前完工？如果有同学拖延，我们可以给出多长的拖延窗口？

最后，我们讨论一个最贴近现实的问题：每个人都不是一个理性人，都会有自身的情绪和状态，换言之，现实中的「小组协作」远不止把任务拆开、塞给不同的人那么简单——同学们各有各的生活节奏、精力上限与情绪波动。如果某位同学被迫熬夜写代码太久，他的满意度会迅速下降；反过来，如果分配稍显轻松，他又可能乐在其中。于是我们面对的真正难题是：**如何在满足总产量要求的同时，把全组的精神状态放到最好？**同时，由于事实上一个人“接盘”另一个人的任务本身就会大大降低任务的效率（这是因为实际上的交接顺序导致的，包括是否能够理解上一个人所做的任务?），并且也会在一定程度上降低组员的满意度，因此在最终的现实情况考虑中，我们否定这一个情况的出现。

三、 我们怎么在规定 DDL 前完成问题?

(一) 问题描述

同一时刻一个小组面临着有 N 个子任务。这个小组共有 M 位组员，不同的组员对不同的问题进行解决所用的时间是不同的。现在需要安排这 M 位组员所解决问题的归属及顺序，使得完成作业平均的时间最小。说明：作业的完成时间是指问题从小组划分完子任务开始到任务完全完成的时间。

(二) 输入输出描述

为了规范算法的输入，我们提出以下标准:

第一行包含两个正整数 M, N，分别表示小组成员的数量与子任务的数量。

接下来 N 行，每行包含 M 个整数。第 $i+1$ 行第 j 个数表示第 j 位小组成员完成第 i 个子问题所需的时间 $T_{i,j}$。

考虑到真实情况，我们可以限定 $2 \leq M \leq 100$，$1 \leq N \leq 60$，$1 \leq T_{i,j} \leq 10^3$。保证小组组员数量一定的情况下（这是因为校内的组队一般较小，但考虑到小组任务的广泛性，在大型工程任务上可能也会适用，因此我们先定一个合理值），合理的时间范围（可以通过规范化规范到这一区间）来进行限制并加以解决。

算法将输出一个最小的时间和对应的方案。

(三) 算法基础

分析问题，我们可以将问题抽象如下:

给定 M 位组员和 N 个子任务，每位组员完成第 i 个子任务需耗时 $T_{i,j}$。在分配子任务时，不仅要决定每个任务由谁完成，还要确定各组员的执行顺序。若第 j 位组员承担了编号为 $i_{j,1}, i_{j,2}, \ldots, i_{j,n_j}$ 的任务，则其第 p 个任务的完成时间为

$$\sum_{q=1}^{p} T_{i_{j,q},j}$$

因此，总完成时间为

$$\sum_{j=1}^{M}\sum_{p=1}^{n_j}\sum_{q=1}^{p} T_{i_{j,q},j} = \sum_{j=1}^{M}\sum_{p=1}^{n_j} p\, T_{i_{j,p},j}.$$

目标是令上述总和最小，所得值除以 N 即为平均完成时间。

最初接到此问题时，我首先考虑:

(1) 直接枚举所有分配和顺序方案，但随着 M, N 增大，组合数爆炸;

(2) 贪心地让最快组员优先处理最快任务，或基于任务时长排序，但此类启发式方法缺乏全局最优保证。

思考到问题的核心是“分配”与“排序”两层决策需要统一度量，才灵光乍现：能否将每个“(组员 j, 排位 p)”都当作一个待匹配位置，并赋予其权重 $p \times T_{i,j}$? 这样，每个任务只需匹配到一个“组员-排位”即可同时确定归属和执行顺序。匹配完毕后总权重即对应总完成时间。

在明确这一建模思路后，最小费用最大流（或带权二分匹配）便是求解此类完美匹配的经典工具，于是最终敲定使用费用流算法。

首先，我们有必要介绍一下最小费用最大流算法，因为这将是我们整个报告一个很重要的出发点。在此之前我们需要介绍一系列前置知识。

1. 费用

在最大流（Max-Flow）中, 我们维护一个「残量网络」Residual Graph, 边上有剩余容量（residual capacity)。在费用流（Min-Cost Flow）中，每条边还带有「单位流量费用」cost。

- **残量网络**：对于每条原始边 $(u \to v)$，增加
  - 正向边 $(u \to v)$：容量 $\mathrm{cap}(u,v)$，费用 $\mathrm{cost}(u,v)$；
  - 反向边 $(v \to u)$：容量 0，费用 $-\mathrm{cost}(u,v)$。

增广过程：在残量网络中找到一条从 s 到 t 的增广路径，瓶颈容量为

$$f = \min_{e \in \text{路径}} \big(e.\mathrm{cap}\big).$$

沿路径推送 f 单位流量，并更新

$$e.\mathrm{cap} \leftarrow e.\mathrm{cap} - f, \quad \mathrm{rev}(e).\mathrm{cap} \leftarrow \mathrm{rev}(e).\mathrm{cap} + f.$$

累积总费用

$$\mathrm{cost} \mathrel{+}= f \times (\text{路径费用之和}).$$

要让费用尽可能低，就要每次选取当前残量网络中 s→t 的「最小费用路径」。由于残量网络可能有负权边（反向边），常用 SPFA 来求解。

2. SPFA 算法

SPFA（Shortest Path Faster Algorithm）是一种求单源最短路径的算法，常用来解决带负权边（但无负权回路）的图的最短路径问题。它是对 Bellman-Ford 算法的一种改进。其核心思路是——“只有被松弛过的节点才有可能再松弛它相邻的边”，因此用一个队列动态维护“待处理的节点”，避免每次都对所有边做无谓检查。

核心思路是：

1. 用一个队列维护可能松弛（relax）的点，初始只有源点入队。
2. 每次从队列取出一个点 u，将其标记为不在队列中；
3. 遍历 u 的所有出边 $(u \to v, w)$，若 $\mathrm{dist}[v] > \mathrm{dist}[u] + w$，则更新 $\mathrm{dist}[v]$ 并记录 $\mathrm{pre}[v] = (u, e)$，若 v 不在队列，则将 v 入队。
4. 重复直到队列为空。

其中：

- **`dist[v]` 数组**：记录从源点 s 到顶点 v 的当前最短路径估计值。
  - 初始化时：`dist[s] = 0`，其它 `dist[v] = +\infty`；
  - 每当发现一条更短的路径 $s \to \cdots \to u \to v$ 时，就做 `dist[v] = dist[u] + w(u,v)`。
- **`pre[v]` 数组**：用来在松弛时记下“从哪个顶点，通过哪条边”到达 v。通常可以拆成两部分：
  1. `pre_v[v]`：记录前驱顶点 u 的编号；
  2. `pre_e[v]`：记录在邻接表 `adj[u]` 中，对应那条从 u 到 v 的边的索引。

我们可以给出他的核心算法逻辑：

Algorithm 1 SPFA(s)

```
初始化 ∀v : dist[v] ← +∞, dist[s] ← 0
队列 Q ← {s}，置 inQ[s] ← true
while Q ≠ ∅ do
    u ← Q.pop()，inQ[u] ← false
    for each 边 e = (u, v, w)（残余容量 > 0） do
        if dist[u] + w < dist[v] then
            dist[v] ← dist[u] + w,  pre[v] ← (u, e)
            if not inQ[v] then
                Q.push(v),  inQ[v] ← true
            end if
        end if
    end for
end whilereturn dist, pre
```

我们给出这一算法的**正确性证明：**

不变量

$$\forall\,(x \to y)\,,\quad dist[y] \;\le\; \min_{\text{路径}s\to\cdots\to x\to y}\Big(dist[x] + w(x,y)\Big).$$

即在任何时刻，$dist[\cdot]$ 都不会高估任意顶点的最短路长度。

松弛次数与路径长度

- 任意一条简单路径 P 从 s 到 v 最多经过 k 条边时，如果沿 P 上每条边都松弛一次，就能将 $dist[v]$ 至少更新到
$$\sum_{e\in P} w(e).$$
- 在无负权回路图中，任意最短路至多包含 $V-1$ 条边。因此，只要对所有“改进机会”做最多 $V-1$ 轮松弛，就能使所有 $dist[v]$ 收敛为真实最短值。
- SPFA 借助队列，将“被上游更新影响”的顶点不断放入队列，触发它们的出边松弛。每条边在其起点距离最终确定后最多被松弛一次，队列空时所有松弛完成。

  接下来说明他的时空复杂度：设图 $G=(V,E)$，其中顶点数 $|V|=V$，边数 $|E|=E$。

1 时间复杂度

1. **成功松弛次数上界**
   对于任意顶点 v，其最短距离 `dist[v]` 最多被更新 $V-1$ 次（最短路至多含 $V-1$ 条边）。因此每条边至多被成功松弛 $V-1$ 次，总代价 $\le (V-1)E = \mathcal{O}(VE)$。
2. **进队／出队次数**
   仅当一次松弛成功且终点 v 不在队列时才入队。故顶点入队次数 $\le V-1$，队列操作总次数亦为 $\mathcal{O}(VE)$。
3. **扫描出边成本**
   每次顶点出队后扫描其全部出边；所有出边被扫描的总次数与出队次数同阶，所以依旧是 $\mathcal{O}(VE)$。

$$\boxed{\text{最坏时间复杂度} = \mathcal{O}(VE)}$$

在随机或稀疏图中，顶点平均进队次数远小于 V，实际运行常接近 $\mathcal{O}(E)$。

2 空间复杂度

邻接表存储	$\mathcal{O}(E)$
距离、标记、前驱数组	$3\mathcal{O}(V) = \mathcal{O}(V)$
队列（最坏容纳V 个顶点）	$\mathcal{O}(V)$

$$\boxed{\text{空间复杂度} = \mathcal{O}(V+E)}$$

但在大多数实际图上，其平均运行时间通常远优于理论上界。

3. 最小费用最大流算法

像网络流一样每个边都有自己的容量，现在每个边除了流量，现在还有一个单位费用，这条边的费用相当于它的单位费用乘上它的流量，我们要保持最大流的同时，还要保持边权 * 最小。

我们采用贪心的思想，以及和 *Dinic* 一样的思路，进行找增广路，每次找到一条从源点到达汇点的路径，增加流量，且该条路径满足使得增加的流量的花费最小，直到无法找到一条从源点到达汇点的路径，算法结束。

由于最大流量有限，每执行一次循环流量都会增加，因此该算法肯定会结束，且同时流量也必定会达到网络的最大流量；同时由于每次都是增加的最小的花费，即当前的最小花费是所得到当前流量 *flow* 时的花费最小值，因此最后的总花费最小。

我们完全可以跑一遍 *SPFA*：在更新的过程中，如果更新成功，那么顺便进行往下一个结点尽可能多地流水，并且把流水的路径记录下来（$pre[v] = u$ 的边），然后跑完 SPFA 后，顺着之前记录的路径从汇点溯回到原点，并且进行增广路，最小的总费用就是累计的费用。

我们来说明其正确性：如果我们有一个容量网络，那他的最大流一定是一个定值（即使是有多个一样的最大值），所以我们从开始的可行流开始增广时，最终的增广量是一定的，所以为了满足最小费用我们只需要每次找最小费用的增广路即可，最后一定是最大流。

Algorithm 2 MinCostMaxFlow(S, T) ——基于 SPFA 的实现

Input: 残量网络 $\mathcal{G} = (V, E)$, 源点 S, 汇点 T

Output: (maxFlow, minCost)

maxFlow $\leftarrow 0$, minCost $\leftarrow 0$
while SPFA(S, T) 找到了增广路 **do**
  回溯 pre 得到瓶颈 $\Delta \leftarrow \min_{e \in \text{path}} cap_{\text{res}}(e)$
  for each 边 e 在路径上 **do**
    $cap_{\text{res}}(e) \leftarrow cap_{\text{res}}(e) - \Delta$
    $cap_{\text{res}}\big(e^{\text{rev}}\big) \leftarrow cap_{\text{res}}\big(e^{\text{rev}}\big) + \Delta$
  end for
  maxFlow $\leftarrow$ maxFlow $+ \Delta$
  minCost $\leftarrow$ minCost $+ \Delta \times$ dist$[T]$
end while
return (maxFlow, minCost)

设残量网络中顶点数为 V，有向边数（含反向边）为 E，最大可增流量为 $F_{\max}$。

1 时间复杂度

1. **单次 SPFA** SPFA 在最坏情况下等价于 Bellman–Ford，需至多 $V-1$ 轮松弛，每轮扫描全部 E 条边：$\mathcal{O}(VE)$。

2. **增广次数**算法每找到一条增广路就至少把 *1* 单位流量推送到汇点，故总循环次数不超过最大流 $F_{\max}$。

$$\text{循环次数} \le F_{\max}.$$

3. **总体时间**

$$T_{\max} = F_{\max} \times \mathcal{O}(VE) = \boxed{\mathcal{O}(F_{\max}VE)}.$$

在单位容量网络 ($F_{\max} \le E$) 下，可写作 $\mathcal{O}(VE^2)$；若边容量或流量上界较小，实际运行常远优于理论上界。

2 空间复杂度

邻接表存图	$\mathcal{O}(E)$,
距离 `dist`， `pre`， `inQ`	$3\mathcal{O}(V) = \mathcal{O}(V)$,
队列（最坏同时容纳V 个顶点）	$\mathcal{O}(V)$.

$$S_{\max} = \boxed{\mathcal{O}(V+E)}.$$

(四) 算法解答

我们可以轻松地借助上面的算法解决这一问题，但需要先做一定的分析才能适配。

对于每个组员而言，他在完成第 k 项任务的时候，之前已经完成了 k−1 项任务，所以对于不同的 k，对应的任务等待的时间是不同的，因此我们需要将每个组员拆成 n 个点，分别表示完成第几个任务的他自己，也就是说，完成前一个任务的他和完成这一个任务的他不是一个他。

然而，若直接把“某项任务 $\to$ 某位组员”连一条边，并试图在边权中同时包进 **排队顺序**的信息，会发现边权难以刻画。因此需要先做一个等价的等待时间展开。

设某位组员最终要完成 K 项任务，且依次耗时

$$T_1,\ T_2,\ \ldots,\ T_K.$$

- 当他完成第 1 项任务时，后面 $K-1$ 个任务都在等待；再加上第 1 个任务本身也在等这个任务完成，因而累计等待时间为 $K \times T_1$。
- 完成第 2 项任务时，仍有 $K-1$ 个任务在等待，因而贡献 $(K-1) \times T_2$。
- $\ldots$

以此类推，总等待时间可写成

$$\sum_{i=1}^{K} T_i\,(K-i+1). \tag{1}$$

注意到式 (1) 也可理解为:

$$\underbrace{T_K \times 1}_{\text{倒数第 1 项}} + \underbrace{T_{K-1} \times 2}_{\text{倒数第 2 项}} + \cdots + \underbrace{T_1 \times K}_{\text{倒数第}K\text{项}}$$

——即“**倒数第 i 项任务**对应贡献 $T_{K-i+1} \times i$ 的等待时间”。

由此得到的建图方案

1. 把 **每位组员** j **拆分成** n **个节点** $u_{j,1}, u_{j,2}, \ldots, u_{j,n}$，分别表示他要完成的“倒数第 k 项任务”的位置。
2. 若第 j 位组员完成第 i 项任务耗时 $T(i,j)$，就 **枚举** $k = 1, \ldots, n$，从“任务节点 v_i”向“组员节点 $u_{j,k}$”连一条有向边：
$$v_i \longrightarrow u_{j,k}, \quad \text{容量} = 1, \text{费用} = k \times T(i,j).$$
   - 此时 k 正是“倒数第 k 项任务”中的 k，与式 (1) 中的乘系数一致；
   - 因容量设为 1，确保每个任务只匹配一个组员。
3. 源点 S 向每个任务节点 v_i 连边 $(1, 0)$，每个“组员-任务”节点 $u_{j,k}$ 向汇点 T 连边 $(1, 0)$。
4. 在该网络上跑 **最小费用最大流**：
$$\text{总费用} = \sum(\text{等待时间}) \implies \text{平均等待时间} = \frac{\text{总费用}}{n}.$$

这样，任务排队顺序已自然编码在边权中，问题转化为“把 N 个任务匹配到 $M \times N$ 个 *(***组员***,* **位置***)* 结点，使费用最小”，恰为最小费用最大流（或带权二分匹配）可直接解决的模型。

不难看出，要得到最终的工作方案，结合算法特点，我们可以发现，在 Add() 在加一条正向边的同时，自动加了容量为 0、费用为 $-$w 的反向边。初始时：正向边容量 $= 1$，反向边容量 $= 0$。若某条边被选入匹配并发送了 1 单位流量：正向边容量变成 0，反向边容量变成 1。于是“反向边容量 ==1”就成了“这条匹配被使用过”的可靠标记。

因此，扫描任务节点的出边：

找到反向容量 ==1 的那条，说明该匹配被用。反解出组员 j 与任务 k，存进 assign_car[i]。最后输出各小组成员的作业队列即可。

（五） 算法复杂度

时空复杂度即为费用流的时空复杂度，将节点数和边数带入即可。

1. 网络构建复杂度

- **节点数**：
$$V = 1\ (\text{源点}) \ + \ N\ (\text{任务}) \ + \ M \times N\ (\text{槽位}) \ + \ 1\ (\text{汇点}) = MN + N + 2 = O(MN).$$
- **边数**：
$$E = N \ + \ N \times M \times N \ + \ M \times N = N^2M + MN + N = O(N^2M).$$

2. SPFA 算法复杂度

在每次增广前，调用 SPFA 构建最短费用层次图，其时间复杂度为
$$O(VE) = O\big((MN) \times (N^2M)\big) = O(M^2N^3).$$

3. 最小费用最大流总复杂度

- **增广次数**：总流量为 N（每个任务一条单位流），每次增广推送 1 单位，故总增广次数为 N。
- **总体时间**：
$$O\big(N \times (M^2N^3)\big) = O(M^2N^4).$$

4. 空间复杂度

$$O(E) = O(N^2M), \quad O(V) = O(MN).$$

四、 我们怎么合理安排时序任务？——如果有同学拖延

(一) 问题描述

我们的小组现在依旧有 m 位组员！我们的目标就是把恩师们布置的作业都做完。恩师一天会布置 n 项任务，其中第 i 项的作业量为 p_i，会在第 r_i 时刻被布置出来，并且必须在第 d_i 秒之前将它完成。第 j 名组员完成任务的速度为 s_j，因此如果它单独做完第 i 项任务所需的时间为

$$\frac{p_i}{s_j}.$$

我们小组作业的策略类似上面的分析，具体来说：

1. 在任一时刻，一位组员最多承担一份作业；

2. 在任一时刻，一份作业只能由一个人独立完成（事实上现实中的子任务也是这样的）。

由于作业的 ddl 常常迫在眉睫，为了将它们全部解决，我们需要给定一个特殊的容忍期限，延长各个组员提交子任务的 ddl。将作业的 ddl 延长 T 秒是指所有的作业的截止时间 d_i 变成 $d_i + T$。同时，我们的容忍度通常不高，因此我们希望找到最小的 T，使得可以完成所有的任务。

输入格式：

第一行包含一个整数 K，表示数据组数。

每组数据的第一行包含两个整数 n 和 m，分别表示任务和组员的数量。接下来的 n 行每行包含三个整数 p_i, r_i, d_i。最后 m 行每行包含一个整数 s_j。上述符号含义如题面所述。

输出 K 行，每行包含一个实数，表示你找到的最小的 T。

(二) 算法基础

当我们有多位组员，速度分别为

$$v_1 \geq v_2 \geq \cdots \geq v_m,$$

如果直接把每个人在每个时间段能贡献的工作量 $v_j \times \Delta t$ 都连进流图，就会出现在同一个时间窗口里多条来自不同组员的边，可以无限叠加的局面，无法保证“每个时刻—一组员只能做一份作业”这一要求。

差分算法的核心思路是：

1. 将原始的速度向量

$$(v_1,\, v_2,\, \ldots,\, v_m)$$

2. 分解成若干个“速度增量”：

$$\big(v_1 - v_2,\; v_2 - v_3,\; \ldots,\; v_{m-1} - v_m,\; v_m\big).$$

这样，增量序列相加依然等于原始速度序列的和（线性无损），但每一个“增量”只对应一条“层”边，就能避免不同组员在同一时间段重复计算工作能力。因此，我们在本题中引入了差分算法。

(三) 算法解决

开始我解决这一问题最初的想法就是使用费用流来完成，但是我仔细分析了一下，发现好像根本用不到费用流？因为此处我根本不用再构造两个权值。因此我们可以直接用课上学到的最大流的思路来解决。

当我们面临的核心问题是：我们的小组作业每天需要应对 n 项子任务，第 i 项作业从时刻 r_i 开始，要求在我们组长预期的截止时刻 d_i 之前完成，工作量为 p_i。我们的小组共有 m 名组员，第 j 名组员的加工速度为 s_j，意味着他单独完成工作量为 p 的任务需要

$$\frac{p}{s_j}$$

的时间。为了应对紧迫的截止日期，我们可以为所有任务统一延长一个容忍时间 T，即把每个任务的截止时间变为 $d_i + T$。问题等价于：在允许任务可以被任意切分、且同一时刻每个组员至多处理一个任务、每个任务至多被一人处理的前提下，找出最小的 T，使得所有任务都能在延长后的截止时限内完成。

为了解这个最小 T，我们首先注意到：如果给定了一个候选值 T，可以通过构造一个“最大流网络”来验证在延期 T 的条件下能否做完总量

$$\sum_i p_i$$

的工作。核心思路是将时间离散化、将组员在各时间段的“工作能力”转换成流量，再用最大流判断是否能够推送完所有任务的工作量。时间离散化步骤中，我们把每个任务的开始时刻 r_i 与延长后的截止时刻 $d_i + T$ 收集，排序并去重后得到若干相邻时间区间

$$[\tau_k, \tau_{k+1}),$$

对于每个区间长度

$$\Delta_k = \tau_{k+1} - \tau_k,$$

可以认为在这段时间里系统状态不变——哪些任务可做、哪些任务已过期都是固定的。

如果直接将“第 j 名组员在第 k 段时间能贡献的工作量”$s_j \times \Delta_k$ 建成容量边，就会出现同一个任务节点同时接受多条来自不同组员的边而无法保证“每个时刻一项任务只能由一人执行”。为此，我们引入“差分速度分层”技巧：先将所有速度排序为

$$v_1 \geq v_2 \geq \cdots \geq v_m,$$

再定义增量速度

$$\delta_1 = v_1 - v_2,\ \delta_2 = v_2 - v_3,\ \ldots,\ \delta_{m-1} = v_{m-1} - v_m,\ \delta_m = v_m.$$

这样，$\sum_{j=1}^{m} \delta_j = v_1$，且在任意时间段中，分别开辟 m 条“层边”容量为 $\delta_j \Delta_k$，只要流量要经过第 j 条层边，就恰好表明此时有“恰好 j 名最快的组员”在做工，这种设计隐式地限制了同一时刻一份任务只能被一名组员加工，也保证了同一时刻每名组员不被分配超过自己速度的总工作。

具体地，在给定一个候选延迟 T 后，我们首先对所有任务的开始和结束时刻进行离散化：将每个任务的发布时间 r_i 与延长后的截止时刻 $d_i + T$ 一起收集、排序并去重，得到一系列关键时刻

$$\tau_1 < \tau_2 < \cdots < \tau_K,$$

从而形成若干连续的同质时间段

$$[\tau_k,\ \tau_{k+1}).$$

网络中，我们设置一个源点 S 和一个汇点 T。在源点与每个任务节点 i（共 n 个）之间连一条容量为该任务总工作量 p_i 的边，表示要“发送”完整的工作量；而在每个时间段 k 与每个速度“层”j（共 m 层）对应的位置，我们创建一个中间节点 $u_{k,j}$。对于这个中间节点，我们会做两件事：

1. 从它到汇点 T 连一条容量为

$$\delta_j \left(\tau_{k+1} - \tau_k\right)$$

的边，其中 δ_j 是第 j 名组员与第 $(j+1)$ 名组员速度的差分（最后一层用 v_m），这条边严格限制了在该时间段内“恰好有第 j 名最快的组员”额外能够提供的总加工量；

2. 如果任务 i 的可用区间 $[r_i,\, d_i+T]$ 完全覆盖了该时间段，则从任务节点 i 向 $u_{k,j}$ 连一条容量为

$$v_j\left(\tau_{k+1}-\tau_k\right)$$

的边，其中 v_j 是第 j 名组员的原始速度。这条边表示“在这一段时间里，这名组员可以为该任务贡献的最大工作量”。

这样，一个完整的任务 i 的所有工作量 p_i 必须从源点 S 流进其任务节点，经过若干个“层”节点，最终汇聚到 T。若最大流值恰好达到

$$\sum_i p_i,$$

则说明所有任务都能在各自可用区间内被恰好完成，没有任何超载，也没有任何时刻一项任务被多人重复处理。

为了找到最小的 T，我们在区间

$$\left[0,\ \frac{\sum_i p_i}{\max_j s_j}+1\right]$$

上进行二分搜索。初始时，左端点设为 0，右端点取为“所有工作量都由最快组员连续完成所需的最大时间再加一”，保证可行解在区间内。每次取中点

$$T_{\text{mid}}=\frac{\text{left}+\text{right}}{2},$$

调用上述最大流可行性检查子程序：离散化时间、构建网络、运行 Dinic 算法求最大流并与总工作量 $\sum_i p_i$ 比较。如果可以成功送出足够流量，说明延期 T_{mid} 足够，则将右端点更新为 T_{mid}；否则将左端点更新为 T_{mid}。在左右端点之差小于预设精度 ε 后终止，最终的中点即为最小可行 T。通过这种在对数时间复杂度（关于答案区间／精度）的二分框架下，每次用最大流判定调度可行性，能够高效、精确地求解出满足所有约束的最小延迟。

我们使用学习的 Dicnic 算法来解决这一问题，伪代码如下。

Algorithm 3 Dinic 算法

Input: 有向图 $G=(V,E)$，源点 s，汇点 t

Output: 最大流值 flow

```
procedure DinicMaxFlow
    flow ← 0
    while BFS() do
        初始化 ptr[u] ← 0 对所有 u
        while (δ ← DFS(s, ∞)) > 0 do
            flowδ
        end while
    end while
    return flow
end procedure
```

Algorithm 4 求解最小延迟 T 的主过程

Input: 测试组数 K

for test $= 1$ to K **do**
  读取 n, m
  读取任务列表 $\{(p_i, r_i, d_i)\}_{i=1}^{n}$，令 totalWork $\leftarrow \sum_i p_i$
  读取速度数组 $\{s_j\}_{j=1}^{m}$ 并降序排序
  for $j = 1$ to $m - 1$ **do**
    $\delta_j \leftarrow s_j - s_{j+1}$
  end for
  $\delta_m \leftarrow s_m$
  left $\leftarrow 0$, right $\leftarrow \frac{\text{totalWork}}{s_1} + 1$
  while right $-$ left $> \varepsilon$ **do**
    mid $\leftarrow$ (left + right)/2
    if FEASIBLE(mid) **then**
      right $\leftarrow$ mid
    else
      left $\leftarrow$ mid
    end if
  end while
end for

涉及到的 Feasible 代码如下算法5所示:

(四) 算法复杂度

时间离散化

```
Time[i]   = x_i;
Time[n+i] = y_i + T;
sort(Time+1, Time+2*n+1);
```

- 时间点数：$2n$
- 时间段数：最多 $2n - 1$

节点构成

$$S,\ T,\ \underbrace{1 \sim n}_{\text{任务节点}},\ \underbrace{n+1 \sim n+m}_{\substack{\text{层节点} \\ (2n-1)\times m \text{ 个}}}$$

$$V = 1 + 1 + n + (2n-1)m = O(nm)$$

边的构成

1. 源点到任务：n 条

$$(S \to i), \quad i = 1 \dots n$$

2. 任务到层节点：$n \times (2n-1) \times m = O(n^2 m)$ 条

Algorithm 5 可行性检查 Feasible(T)

Input: 候选延迟 T

构造时间点集 $\{ r_i,\ d_i + T \mid i = 1 \dots n\} \to \{\tau_k\}_{k=1}^{K}$（排序去重）
初始化 Dinic 网络，设源点 S，汇点 T
for $i = 1$ to n **do**
    添加边 $(S \to i)$ 容量 p_i
end for
for $k = 1$ to $K - 1$ **do**
    $\Delta \leftarrow \tau_{k+1} - \tau_k$
    if $\Delta \leq 0$ **then**
        continue
    end if
    for $j = 1$ to m **do**
        $u \leftarrow \text{layerNode}(k, j)$
        添加边 $(u \to T)$ 容量 $\delta_j \cdot \Delta$
        for $i = 1$ to n **do**
            if $r_i \leq \tau_k$ 且 $\tau_{k+1} \leq d_i + T$ **then**
                添加边 $(i \to u)$ 容量 $s_j \cdot \Delta$
            end if
        end for
    end for
end for
maxf $\leftarrow$ Dinic.maxflow(S, T)
return $\big(\text{maxf} \geq \text{totalWork}\big)$

3. 层节点到汇点：$(2n-1)\times m = O(nm)$ 条

$$E = n + O(n^2m) + O(nm) = O(n^2m)$$

时间复杂度分析

单次 Dinic Dinic 算法最坏时间复杂度：$O(V^2E)$。

$$V = O(nm),\quad E = O(n^2m) \quad\Longrightarrow\quad O\big((nm)^2\cdot n^2m\big) = O(n^4m^3).$$

网络构建

$$O(n\log n)\ \text{(排序)}\ +\ O(n^2m)\ \text{(加边)} = O(n^2m).$$

二分搜索

$$O\big(\log\frac{1}{\varepsilon}\big) \approx O(\log 10^6) \approx 20.$$

总体时间复杂度

$$O\Big(\log\frac{1}{\varepsilon}\ \times\ (n^2m + n^4m^3)\Big) = O(n^4m^3).$$

空间复杂度分析

- 邻接表：$O(E) = O(n^2m)$
- 图结构数组：$O(V) = O(nm)$
- 其他辅助数组：$O(n+m)$

$$\text{空间总量} = O(n^2m).$$

五、 我们怎么面对小组组员的情绪？——最现实的情况

(一) 问题描述

你的小组接到了老师最新布置的一个任务（太好了!），假定你已经将这一大任务分成了若干个子任务。任务要求你们小组完成 n 项子任务（编号 $1\sim n$），其中第 i 类任务共需要 C_i 个单位工作量。小组共有 m 名组员（编号 $1\sim m$），不同组员会做的任务种类各不相同。一个单位工作量必须由 **同一**名组员独立完成，不能拆分给多人。

我们用一张 $m\times n$ 的 0/1 矩阵 **A** 描述组员能力：

$$A_{i,j} = \begin{cases} 1, & \text{组员}i\ \text{能做该任务}j, \\ 0, & \text{否则。} \end{cases}$$

行和列分别被编号为 1-m 和 1-n。

如果作为组长的你分配了过多工作给一名组员，这名组员会变得不高兴。我们用愤怒值来描述某名组员的心情状态。愤怒值越高，表示这名组员心情越不爽，愤怒值越低，表示这名组员心情越愉快。组员的愤怒值与他被安排完成的工作量存在某函数关系，鉴于组员们的承受能力不同，不同组员之间的函数关系也是有所区别的。

对于组员 i，他的愤怒值与工作量之间的函数是一个 S_i+1 段的分段函数。当他完成第 $1\sim T_{i,1}$ 类任务时，每份工作量会使他的愤怒值增加 $W_{i,1}$，当他完成第 $T_{i,1}+1\sim T_{i,2}$ 个工作量时，每个工作量会使他的愤

怒值增加 $W_{i,2}$ ……为描述方便，设 $T_{i,0}=0, T_{i,s_{i+1}}=+\infty$，那么当他应对第 $T_{i,j-1}+1 \sim T_{i,j}$ 类任务时，每份工作量会使他的愤怒值增加 $W_{i,j}$，$1 \le j \le S_i+1$。

目标是制定出一个任务的分配方案，使得我们能够向老师提交一份完整的作业，并且为了维持组员之间友好的关系，需要所有组员的愤怒值之和最小。你需要输出愤怒值和对应的安排策略。

规范输入格式：第一行包含两个正整数 m 和 n，分别表示组员数量和任务的种类数；

第二行包含 n 个正整数，第 i 个正整数为 C_i；

以下 m 行每行 n 个整数描述矩阵 A；

下面 m 个部分，第 i 部分描述组员 i 的愤怒值与任务工作量数量的函数关系。每一部分由三行组成：第一行为一个非负整数 S_i，第二行包含 S_i 个正整数，其中第 j 个正整数为 $T_{i,j}$，如果 $S_i=0$ 那么输入将不会留空行（即这一部分只由两行组成）。第三行包含 S_i+1 个正整数，其中第 j 个正整数为 $W_{i,j}$。

(二) 算法解答

这道题看似和先前第一题使用的费用流很相似，但显然笔者将其作为最后的压轴问题显然不会这么简单(误)，我们可以发现其愤怒值是一个分段函数！这就意味着和先前的固定 cost 不能采用一模一样的策略。

当我们拿到这样一个任务分配问题时，首先要明确：总共有 n 类子任务，第 j 类需要完成 C_j 个单位工作量；共有 m 名组员，每个人能完成的子任务类型不尽相同，由一个 $m \times n$ 的二值矩阵 A 给出。矩阵中若 $A_{i,j}=1$，则组员 i 可以承担第 j 类任务，否则这类任务对他不可用。每个单位的工作量必须由同一人完整完成，不能拆散到多人。这一基本约束保证了任务分配的可行性和互斥性。

更为特殊的是，每位组员 i 的“愤怒值”并非线性，而是一个由 S_i+1 段分段函数决定的凸增函数：当他承担第 1 到 $T_{i,1}$ 单位工作时，每单位额外付出的愤怒成本是 $W_{i,1}$；当他承担第 $T_{i,1}+1$ 到 $T_{i,2}$ 单位工作时，每单位成本上升为 $W_{i,2}$，……直到最后一段无限延伸，每单位成本为 W_{i,S_i+1}。我们的目标是在满足所有工作量都被分配（即总流量达到 $\sum_j C_j$）的前提下，使得所有组员的累积愤怒值之和最小。这个“分段定价”的特性，直接决定了我们必须在建模时将“费用”随流量变化的情况精确展现出来。

我们最后采用的做法是将该问题转化为带费用的网络流问题——最小费用最大流（MCMF），就像上面讨论的那样。我们先把每类子任务看作一个“任务节点”U_j，再引入一个超级源点 s，用一条容量为 C_j、费用为 0 的边把 s 和 U_j 连起来，保证恰好要将 C_j 单位的“作业量”从源头“发送”到这个节点。然后，对于每个组员 i，在网络中为其建立一个“组员节点”V_i。如果 $A_{i,j}=1$，则在任务节点 U_j 到组员节点 V_i 之间连一条容量足够大（可取 $\sum_j C_j$）且费用为 0 的边，表示“任务 j 的任意单位都可以流向组员 i”，不受数量上限的限制，只受能力许可的约束。

最关键的地方就在于如何将组员的分段愤怒函数“搬”到这张流图中。我们对每个组员 i 到汇点 t 的那条“汇聚边”进行“拆边”操作：将其分割成 S_i+1 条平行边，分别对应原函数的每一段。第 k 条平行边的容量记为

$$\Delta T_{i,k} \;=\; T_{i,k}-T_{i,k-1},$$

其中 $T_{i,0}=0$、$T_{i,S_i+1}=\sum_j C_j$，费用为 $W_{i,k}$。这样，如果组员 i 接受了 x 单位工作，网络就会先让最多 $T_{i,1}$ 单位的流量经过第一条低费边，再让余下的 $\min(x-T_{i,1},\, T_{i,2}-T_{i,1})$ 单位流量经过第二条边，以此类推。每条边的“流量 $\times$ 费用”累计之和，正好还原了原问题中分段累加的愤怒值。

有了以上建图，剩下的工作就是在该带费用网络中求最大流（$\sum_j C_j$）且费用最小。这一步我们通常用 SPFA + 带距离标号的 DFS 方案来实现：先用 SPFA 在当前残量网络中以“费用”作为边权，构建从源点到汇点的最短路层次图；然后在该层次图上用 DFS 尽可能多地推阻塞流，并将每单位推送的流量乘以当前最短路长度累加到全局费用上。重复 SPFA 和 DFS 直到再也无法在残量网络中找到增广的最短路径为止。此时全局流量必然达到 $\sum_j C_j$，且对应的总代价即是最小化的组员愤怒值总和，恰为我们所求。

通过这样一套完整的建模与 MCMF 算法，我们既严格地满足了“每人只能做自己能做的任务”“每单位工作不能拆分”“分段定价”的所有约束，又能高效地在 $O(\text{flow} \times E)$（或更优情况下）的时间内，计算出最

优分配方案。

SPFA 算法的伪代码在第一题分析的时候就已经给出，详见算法 1。我们给出解决问题的主要伪代码。

Algorithm 6 核心伪代码

```
1: procedure SOLVE
2:     read m, n
3:     totalWork ← 0
4:     for j = 1 to n do
5:         read C[j]; totalWorkC[j]
6:     end for
7:     source ← n + m + 1, sink ← n + m + 2
8:     for j = 1 to n do
9:         addEdge(source, j, C[j], 0)
10:        addEdge(j, source, 0, 0)
11:    end for
12:    for i = 1 to m do
13:        for j = 1 to n do
14:            read A[i][j]
15:            if A[i][j] = 1 then
16:                addEdge(j, n + i, ∞, 0)
17:                addEdge(n + i, j, 0, 0)
18:            end if
19:        end for
20:        read S_i
21:        T_i[0] ← 0, T_i[S_i + 1] ← totalWork
22:        for k = 1 to S_i do
23:            read T_i[k]
24:        end for
25:        for k = 1 to S_i + 1 do
26:            read W_i[k]
27:            addEdge(n + i, sink, T_i[k] − T_i[k − 1], W_i[k])
28:            addEdge(sink, n + i, 0, −W_i[k])
29:        end for
30:    end for
31:    while SPFA() do
32:        for v = 1 to sink do
33:            cur[v] ← head[v]; visited[v] ← false
34:        end for
35:        sendFlow(source, ∞)
36:    end while
37: end procedure
```

(三) 算法复杂度

节点数

$$V = n\ (\text{任务})\ +\ m\ (\text{组员})\ +\ 2\ (\text{源点}+\text{汇点}) = O(n+m).$$

边数

$$E = \underbrace{n}_{\text{源}\to\text{任务}} + \underbrace{n\times m}_{\text{任务}\to\text{组员}} + \underbrace{\sum_{i=1}^{m}(s_i+1)}_{\text{组员}\to\text{汇点}} = O(nm+S),$$

其中 $S=\sum_i(s_i+1)$。

时间复杂度

1. **SPFA**：每次构建最短费用路径，最坏 $O(VE)=O((n+m)\,(nm+S))$。
2. **DFS 增广**：每次沿零费用边尽量推流，$O(E)=O(nm+S)$。
3. **增广轮数**：最坏每次推送 1 单位，总流量 totalDemand，故最多 totalDemand 轮。

综合得

$$\text{总时间}\ =\ O\big(\text{totalDemand}\times(n+m)\times(nm+S)\big).$$

空间复杂度

- 邻接表存储：$O(E)=O(nm+S)$。
- SPFA 临时数组：$O(V)=O(n+m)$。
- 能力矩阵及需求数组：$O(nm+n)$。

因此总体空间复杂度为

$$O(nm+S).$$

六、 总结

本实验报告通过三个递进的问题深入探讨了小组作业分配的算法解决方案，成功将现实的小组作业分配问题转化为网络流模型，特别是将任务执行顺序通过节点拆分技术巧妙地编码到网络结构中。系统性地应用了 SPFA、最小费用最大流和 Dinic 算法，展现了这些经典算法在实际问题中的威力。在时序调度问题中引入差分速度分层技术，有效解决了多资源并发约束的建模难题；在情绪建模中通过拆边技术实现了分段函数的精确表达。

提供了一套完整的、可操作的小组作业分配解决方案，从简单到复杂层层递进，适应不同的实际需求。虽然理论复杂度较高，但对于实际的大学生小组作业小组规模（M<<100, N<<60），算法在合理时间内可解。符合我们算法的预期目标。

总体而言，本研究成功地将理论算法与实际问题相结合，为复杂的多约束优化问题提供了系统性的解决思路，具有重要的实践意义。

希望真的能对大学生小组作业的完成起到促进作用！在本次实验中，我也对网络流和费用流算法有了深刻的理解！