基于有向无环图最小路径覆盖算法的排班优化策略研究

摘要

本文针对有复杂约束的城市轨道交通乘务人员调度问题，为车次依赖情况建立了加权有向无环图（DAG）模型，使用两种不同的算法求解出最佳调度方案，并定义了一些指标对方案的合理性进行评价。

对于全体最小不可行子链的查找问题，首先通过将任务看作结点、任务之间能够依次执行的关系看作有向边，建立了包含所有可行调度的有向无环图模型。对得到的图进行拓扑排序，按顶点的拓扑序逐个进行深度优先搜索，并在搜索时对符合条件的子链进行筛选，即可找到所有的最小不可行任务链。将算法应用在某个工作日内 242 条车次的数据中，最后得到了 720 条最小不可行子链。经过理论分析和样例检测，肯定了算法不重不漏的正确性。

对于使任务链数量最少的任务调度问题，设计了分两步进行的算法。首先不考虑约束条件，使用 Dinic 算法求解有向图上有约束的最小路径覆盖，将得到的 37 条任务链作为不满足约束的初步调度；然后基于“截断”和“拼接”两种原则对初步调度中不满足约束的子链进行局部修改，在保证总数尽可能少的前提下得到了较优的调度方案，由 41 条任务链组成。得到结果之后，对算法的数据依赖性质进行了分析，指出其不足。

对于均衡化、合理化的任务调度问题，首先定义了资源利用率、休息时间公平性、时间安排合理性三个指标，作为判定调度方案优劣的标准。然后，分别使用“缩点”的预处理技巧和基于随机的 Las Vegas 算法改进了原算法的两个部分，既减轻了休息时间在某条任务链中的分布不均匀的情况，又缩小了查找最优方案的范围，使找到最优解的概率大大提高。使用此算法进行 500 次随机生成，并在第 437 次迭代找到了符合题目要求的方案，由 42 条任务链组成。

计算改进前后的算法得到的方案的相关指标，得到原方案 2 全体司机空闲时间和工作时长的变异系数以及休息时间在任务链中分布的均匀程度，分别是 0.633，0.185 和 1916.850。而改进后方案的相关指标降低为 0.400，0.140 和 1768.110，结合相关图表，证明了改进后算法的优势。

最后我们对模型的优缺点进行了评价，并提出了未来的改进方案。

关键字：任务调度　有向无环图　最小路径覆盖　Dinic 算法　Las Vegas 算法

一、 问题重述

1.1 问题背景

一般的城市地铁系统由多条线路组成，而一条线路通常由固定的乘务人员班组负责。将一条线路以换乘点为界限分成几个值乘片段（车次），司机在执行完一趟车次之后，将换乘到另一趟车次继续执行任务。这样，每位列车司机当天的任务将是以值乘片段为单位串联起来的任务链，对司机的排班调度实质上就是将所有车次合理地划分为几条任务链的过程。

在实际情况下，由于城市轨道交通车次数量多、运行间隔密、换乘困难、法规条例严格等特点，对乘务人员的排班调度存在一系列复杂的约束。为方便分析，将原题目中所给的约束条件重新总结为以下 5 种：

1. **连接约束：**车次只有在衔接地点一致、时序关系正确，并针对反向换乘预留出换乘时间时才能在任务链中相邻。

2. **疲劳约束：**不允许出现驾驶时长总和超过 2 小时的连续几个任务，除非任务之间存在一次换乘时间超过 20 分钟，称为**小休**。

3. **工作日约束：**每条任务链的终止时间减去起始时间应当小于 8 小时。

4. **里程约束：**每条任务链的所有任务里程之和应当小于 200km。

5. **偏好约束：**从 PB 站点开始的任务链数量应当大于从 PA 站点开始的任务链数量的 2 倍。

满足连接约束的任务的排列称为**任务链**，满足疲劳约束的任务链称为**可行任务链**。如何在这些约束下得到司机值乘安排的评估指标及其最优方案，是地铁管理与运营需要解决的重要问题。

1.2 问题要求

问题一定义了**最小不可行任务链**的概念，即去掉任一端点都将成为可行任务链的不可行任务链。要求拼接附件 1 中给出的任务，找出所有的最小不可行任务链并输出。

问题二要求以最小化车次任务链数量为优化目标，建立该地铁运营公司的列车司机排班模型，并进行求解。

问题三要求自行定义衡量指标，分析第 2 问找到的最优列车司机排班方案在任务均衡性、合理性等方面还有哪些不足，并通过可视化作图形式解释选择原因。根据数据分析的结果，从模型或算法角度提出可行的改进方案。

二、 问题分析

在问题分析部分，提到的一些术语参见1.1问题背景部分。

2.1 问题一的分析

在附件 1 给出的数据中，每个车次都表示一个任务。将任务看作图论模型中的一个结点，可以将任务链理解为一条有向路径；将其推广至囊括所有任务的图结构，就可以使用深度优先搜索等图相关的算法来实现目标。事实上，已有很多成熟的研究通过建立图论模型来分析有简单约束的任务调度问题，例如文献 [1]。一般方法是将任务看作图中的结点，将任务的先后依赖关系看作有向边，将原题目转化为**有向无环图**（DAG）相关的问题。

而在本题中，由于连接约束直接决定了两个任务能否在任务链中相邻，因此选择将其作为两结点之间有向边存在的依据。建立有向无环图之后，对已建立的图模型进行**拓扑排序**，再通过**深度优先搜索**遍历所有子链。在遍历的过程中，判断每一条可能存在的最小不可行任务链并输出即可。

2.2 问题二的分析

根据问题的要求，总结得出如下性质：

1. 任务不允许任何延迟，不能通过推迟某个任务来处理冲突。
2. 符合题意的任务链需要满足一些局部或全局的复杂的约束。
3. 不限制结果中任务链的总数，只需要尽量少即可。

问题二要求将任务连接成一些满足所有约束的任务链，使得任务链总数尽可能少。而上面总结的特性使得本问题不同于文献 [1] 中提到的一般的任务调度问题，也就不能使用一般的优化模型解决。基于问题一建立的 DAG 模型，问题二可以转化为求 DAG 的有约束最小路径覆盖（MPC）问题，而对于无约束 MPC 问题，文献 [2] 已经给出了解决方案。

为了得到一条满足约束的任务链，有截断和拼接两种做法，通过这些方式对文献 [2] 中的算法进行改进，得到适合本题的调度方案。

2.3 问题三的分析

问题三对任务链总数的限制放宽，希望得到的结果更加合理与公平。这分为两步解决，首先，定义一些指标来评估第 2 问得到的方案并可视化指标。

1. 对于系统的效率，可以计算司机有多少不必要的休息时间，把这些时间看作资源的浪费；

2. 对于任务的均衡性，可以计算出每位司机的任务量并求其方差；

3. 对于任务的合理性，可以针对每位司机的时间安排，寻找是否存在“工作半小时，休息一小时”的异常情况。

其次，需要从算法角度改进问题 2 得到的方案。以定义的这些评估指标连同任务链总数为优化目标，以全部 5 条约束作为约束条件，使用缩点算法生成新的 DAG 以及使用 **Las Vegas 算法**在解空间中随机搜索，输出此优化问题的最优结果。为了体现出算法的改进，还要对两种方案的指标进行对比分析。

三、 模型假设

1. 行车不会出现任何延误和提前，严格按照附件 1 中的起止时刻运行。

2. 不考虑其他线路列车的停靠对换乘点运作的影响。

3. 正向换乘所需的时间可任意少，几乎可以视为乘务人员下车之后马上能够开始执行下一个任务。

4. 在任务之间，单次低于 20 分钟的换乘时间对驾驶疲劳没有影响，这意味着下一段任务开始后，驾驶疲劳会继续积累；而超过 20 分钟的小休会清空乘务人员的疲劳程度。

四、 符号说明

符号	含义
G	原问题任务及其关系之间抽象出的有向无环图
$G^{(1)}$	在 G 的基础上仅保留所有时间跨度小于 20 分钟的边的 DAG
$G^{(2)}$	在 G 的基础上仅保留所有时间跨度不小于 20 分钟的边的 DAG
$pn(G)$	有向无环图 G 的最小路径覆盖数
$s(v)$	结点 v 对应的任务的接车时间
$t(v)$	结点 v 对应的任务的下车时间
$p(v)$	结点 v 对应的任务的接车地点
$q(v)$	结点 v 对应的任务的下车地点
$d(v)$	结点 v 对应的任务的车次流向
$w(v)$	结点 v 的权重
$x(v)$	结点 v 对应的任务的里程
$c(u,v)$	G 中边 (u,v) 的权重

这里只列出论文各部分通用符号，个别使用不频繁的符号会在首次引用时会进行说明。

五、 模型建立与求解

5.1 问题的数学抽象

5.1.1 为轮值安排定义时间轴

以一天的 00:00 为时间轴原点，将每一天中的每个时刻以秒为单位，标在时间轴中，在后文仅用单位为秒的时间来指代正常的“时: 分: 秒”格式的时刻。

5.1.2 任务之间的关系抽象为图结构

根据问题分析，将每个任务看作有向图中的结点，当满足连接约束时两个结点之间连一条有向边，设任务数为 n，构造一个有向图 $G=(V,E)$，其中 $V=\{v_1,\cdots,v_n\}$, 每个顶点是一个六元组 $v_i=(s_i,t_i,p_i,q_i,d_i,w_i,x_i)$，其中:

1. 结点 v_i 表示任务 i 与车次 i 是一一对应的;
2. s_i 为任务 i 的接车时间，t_i 为任务 i 的下车时间，两个时间单位为秒且为整数;

3. p_i 为任务 i 的接车地点，q_i 为任务 i 的下车地点，为了方便，将站点 PA，PB，PC 分别记作 $0, 1, 2$，为了方便叙述，会采用站点 0 来指代站点 PA，于是 $p_i, q_i \in \{0, 1, 2\}$；

4. d_i 为任务 i 的车次流向，$d_i = 1$ 表示车次流向为正向，即 $p_i < q_i$, $d_i = 2$ 表示反向，即 $p_i > q_i$. 因此 $d_i \in \{1, 2\}$；

5. w_i 为结点 v_i 的权重，满足 $w_i = t_i - s_i$；

6. x_i 表示任务从站点 p_i 到站点 q_i 跨越的里程，单位为 km.

为了书写方便，对于某一个指标（比如 v_i 的接车时间 s_i ），也可以用类似于函数的写法来指代，比如 $s(v_i) = s_i$ 也表示结点 v_i 或者使任务 i 的接车时间。

根据连接约束，需要判断有向边 $(v_i, v_j)(i \neq j)$ 是否在边集 E 中，$(v_i, v_j) \in E$ 当且仅当：

1. 满足空间约束，即 $q_i = p_j$，保证衔接地点一致。

2. 满足时间约束，即

$$\begin{cases} t_i < s_j, & d_i = d_j, \\ t_j + 600 < s_j, & d_i \neq d_j, \end{cases} \tag{1}$$

保证当车次流向一致时，后一个任务的接车时间晚于前一个任务的下车时间；当车次流向不一致时，除满足前一种情况的约束以外，还需留出 10 分钟的换乘时间。

除此之外，对于每一条边，我们定义边权 $c : E \to \{0, 1\}$，满足对 $(u, v) \in E$，有

$$c(u, v) = \begin{cases} 0, & s(v) - t(u) < 1200 \\ 1, & s(v) - t(u) \geq 1200 \end{cases} \tag{2}$$

即我们用 0 和 1 来标注出两个任务之间的时间差距是否超过 20 分钟，以此来判断该两个任务之间是否可以进行小休。

5.1.3 司机排班安排抽象为路径覆盖问题

构造出有向图 G 之后，根据任务的时间先后顺序，可以断言 G 是一个有向无环图（DAG），我们将在这样一个有向无环图中进行问题的研究与求解。将所有任务及其之间的关系抽象为有向无环图之后，任务链随即可以抽象为 G 中的有向路径 $P = [u_1, \cdots, u_k]$，那么乘务人员排班的安排就是将有向无环图 G 分解成若干个不相交的路径 $P_1, \cdots, P_N$，这里的不相交指的是，对任意 $1 \leq i, j \leq N \wedge i \neq j$ 以及任意顶点 $u \in P_i, v \in P_j$，都有 $u \neq v$，同时每一条路径 P 要满足剩下的约束，这些约束写成数学形式如下：

1. **疲劳约束：**设 P 中边权为 1 的边将 P 断开成了若干子路径 $P^{(1)}, \cdots, P^{(k)}$，需要满足

$$\max_{1 \leq i \leq k} \sum_{u \in P^{(i)}} w(u) \leq 7200 \tag{3}$$

2. **工作日约束：**设 P 的起点和终点分别为 u_s, u_e，则 $t(u_e) - s(u_s) \leq 28800$.

3. **里程约束：** $\sum_{u \in P} x(u) \leq 200$.

4. **偏好约束：**设 $P_1, \cdots, P_N$ 的起点分别为 $u_1, \cdots, u_N$，则

$$\frac{\#\{p(u_i) = 0 : 1 \leq i \leq N\}}{\#\{p(u_i) = 1 : 1 \leq i \leq N\}} \leq \frac{1}{2} \tag{4}$$

其中 $\#A$ 表示集合 A 的元素数量，分子、分母分别表示以 PA、PB 为起点的路径数目。

当不考虑上面的 4 条约束时，将有向无环图分解为若干不相交的路径的问题被称为有向无环图的路径覆盖（Path Cover）问题，问题要求最小化 N，于是需要求解的是一个有约束的 MPC 问题。在下面我们会同时使用任务链和路径来进行描述，即任务链和路径的含义是等价的。

5.2 问题一的求解

根据最小不可行任务链的定义，可以得到如下性质：

1. 最小不可行链不存在不可行的真子链。否则，去掉某个端点会使任务链仍然不可行，这与最小性矛盾。

2. 最小不可行链中不存在小休。否则在所有小休处断开任务链，一定会得到不可行的真子链。

根据上述性质，考虑使用深度优先搜索（DFS）来搜索以每一个任务为起点的最小不可行任务链，当该起点的搜索完毕后删掉该点，因此按照顶点的拓扑序来搜索。每一次搜索记录当前所经过的路径的所有任务的总驾驶时间，当总驾驶时间超过 2h 时，得到了一条不可行子链，此时再判断从起始任务开始去除起始任务是否会导致任务链变得可行，如果不会导致变得可行说明该不可行子链不是最小，终止该边的搜索，反之则让答案加一，核心算法用伪代码进行表示，其中算法 1 为拓扑排序的伪代码，算法 2 为 DFS 的伪代码。

算法 1: 拓扑排序：TopologySort

输入: 图 $G = (V, E)$

输出: 顶点的有序序列 Q

```
Q := ∅;
处理出每个顶点的入度 deg(v_1), deg(v_2), ..., deg(v_n);
while V ≠ ∅ do
    选择度数最小的点 u;
    将 u 添加到有序表 Q 的末尾;
    将 u 从 V 中删除并更新 E 以及每个点的入度;
```

该方法枚举了以每个顶点为起点的所有最小不可行子链，保证不会遗漏，同时判断移除起点是否会导致链变得可行来确认搜索到的不可行子链是否是最小的，保证了不会重复，同时题目中列举的两个最小不可行链的例子也在本方法找到的结果中，因此得到的结果应该是正确的。最后得到的结果为 720 条，在附件 8 中，其中只有长度为 3 的路径和长度为 4 的路径，这也符合预期。

算法 2: 深度优先搜索：DFS

输入: 图 $G=(V,E)$，拓扑排序得到的有序序列 Q，当前点 v，本次搜索的起点 `start`，到达当前点 v 之前的路径总权 T

输出: 以 `start` 为起点的最小不可行子链数目 `ans` 及其具体路径

```
if 当前工作总时间（路径的总点权）> 7200 then
    if 当前的不可行路径移除起点start 后成为可行路径 then
        ans := ans + 1;
    return;
T := T + w(v);
for v 的所有邻接顶点u do
    记录的路径增加顶点 u;
    DFS(G, Q, u, start, T);
```

5.3 问题二的求解

在前面已经将问题二转化为了有向无环图的有约束的最小路径覆盖问题，不管有无约束，排班安排都是一个路径覆盖，由于对任意有向无环图 $G=(V,E)$，设 G 的最小路径覆盖数目为 $pn(G)$，文献 [3] 给出了其计算公式：

$$pn(G)=\sum_{v\in V}\max\{0,\deg_{-}(v)-\deg_{+}(v)\} \tag{5}$$

其中 $\deg_{-}(v)$ 表示顶点 v 的入度，$\deg_{+}(v)$ 表示顶点 v 的入度。于是设满足约束的最小路径覆盖数目 $C_{\min}$，那么 $pn(G)$ 是它的下界，即一定有

$$C_{\min}\geq pn(G) \tag{6}$$

同时，MPC 问题可以转化为二分图匹配问题，使用匈牙利算法或者最大流 Dinic 算法在多项式时间复杂度内就能得出一张有向无环图的最小路径覆盖数目 $pn(G)$ 及一种方案。为了充分利用已有的算法，我们决定使用 Dinic 算法求出问题的有向无环图 G 的最小路径覆盖数目及其方案，然后在这个方案的基础上加以改进，使其满足所有的约束，这概括为流程图1。

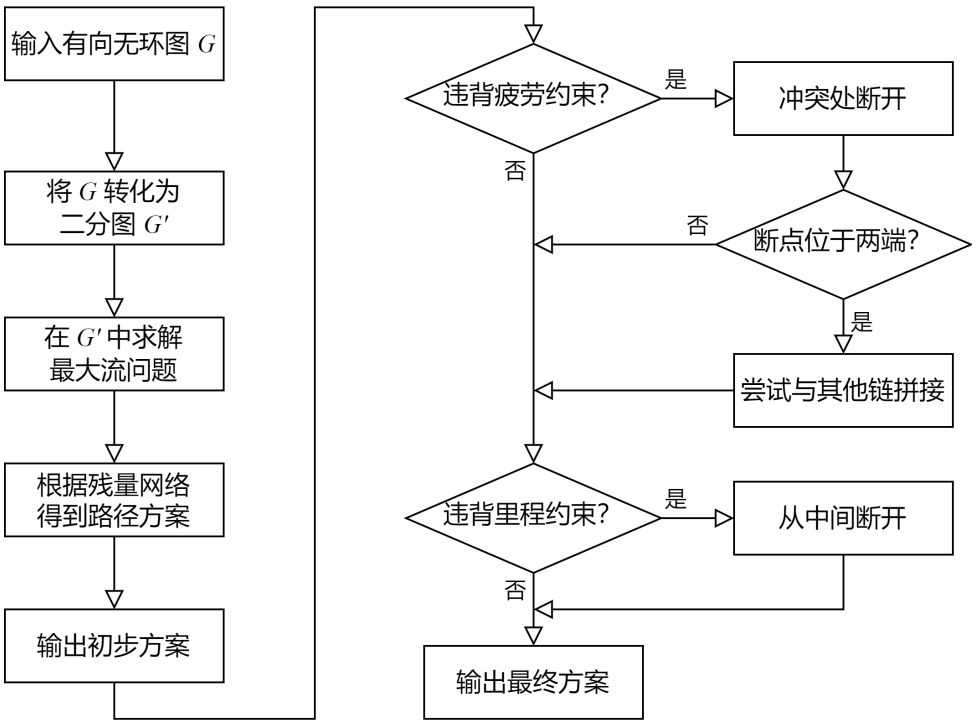


图 1　解决问题二的流程图

5.3.1 使用 Dinic 算法得到初步方案

对于有向无环图 $G=(V,E)$，将 V 中的每个点拆成两部分，分别作为二分图的左侧顶点和右侧顶点，两侧顶点连边当且仅当该边在 G 中，那么求出二分图的最大匹配时 M 时，M 中的每条边只使其左端点的出度和右端点的入度减少了 1，这保证了我们将匹配 M 映射到 G 中时不会出现路径重叠的情况；其次在 G 中，每增加一条匹配边就有两条路径被合并为一条路径，因此由归纳可得**最小路径覆盖数 = 顶点个数 − 最大匹配数**，即 $pn(G)=|V(G)|-|M|$.

现在将构造二分图的过程形式化：以 G 构造二分图 $G'=(V_1\cup V_2,E')$, 其中 $V_1=V$, 为了区分二分图的两部分点, 令 $V_2=\{v_{n+1},\cdots,v_{2n}\}$, 相当于把 v_i 拆成了二分图中的 v_i,v_{i+n} 两个点, 且对于 $v_i\in V_1,v_j\in V_2(1\le i\le n,n+1\le j\le 2n)$, $(v_i,v_j)\in E'$ 当且仅当 $(v_i,v_{j-n})\in E$.

求解二分图最大匹配可以使用匈牙利算法或者 Dinic 算法，在这里我们将其转化为最大流问题使用 Dinic 算法：在 G' 中增加超级源点 s 和超级汇点 t，其中 s 与 V_1 中所有的顶点相连，V_2 中的顶点与 t 相连，并使所有有向边的容量为 1，可行流为整数，记 G' 增加源点和汇点后的流网络为 $G_f=(V_f,E_f)$，它的流函数为 f，求解该流量网络的最大流即可得到原二分图的最大匹配数目，算法3为 Dinic 算法的伪代码。

算法 3 得到了最大流流经 G_f 后的残量网络 G_r，根据残量网络 G_r 便可得到原有向无环图 G 的最小路径覆盖方案。最终，得到 37 条可行链，也就意味着 $pn(G)=37$ 且

算法 3: Dinic 算法：Dinic

输入: 流网络 $G_f = (V_f, E_f)$，网络的流函数 f

输出: 最大流 $f_{\max}$ 求解最大流后的残量网络 G_r

1 初始化最大流 $f_{\max} := 0$;
2 构造辅助网络 $N_A(f)$;
3 **if** $N_A(f)$ 中不存在从s 到t 的可行流 **then**
4 根据 $f_{\max}$ 求解残量网络 G_r;
5 **return** G_r;
6 初始化 $N_A(f)$ 的极大流 g 为零;
7 **if** $N_A(f)$ 中存在顶点v 使得通过它的最大流量为零 **then**
8 **if** v 为源点s 或者汇点t **then**
9 转到第 20 行;
10 **else**
11 在 V_f 中删去 v 及其相连的边并更新 E_f;
12 找到结点 k，满足在所有的顶点中流经 k 的最大流量最小，从 k 开始将流经 k 的最大流量推送到汇点和源点，并加到 g 上;
13 在 V_f 中删去 k 及其相连的边并更新 E_f;
14 **for** $N_A(f)$ 的所有边 e **do**
15 **if** $g(e)$ 等于$N_A(f)$ 中边 e 的容量 **then**
16 删去边 e 并更新 $N_A(f)$ 的边集;
17 **else**
18 $N_A(f)$ 中边 e 的容量减去 $g(e)$;
19 转到第 7 行 ;
20 $f_{\max} := f_{\max} + g$;
21 转到第 2 行;

$C_{\min} \geq 37$，这些链是只考虑连接约束的初步方案。

5.3.2 通过截断-拼接方法改良初步方案

考虑全部约束后，初步方案中的部分任务链产生了冲突之处，如表1所示。为了修改任务链以得到满足约束的调度方案，针对每条任务链违反的约束不同，有截断和拼接两种做法：

1. **截断**：对于不满足疲劳约束的任务链，找到导致疲劳的位置进行截断处理：如果

表 1　违反约束的任务链

编号	内容	违反的约束
1	P001 P041 P082 P124 P157 P196 P230	工作日约束
2	P002 P037 P061 P108 P141 P172 P208 P241	里程约束
4	P004 ***P039 P070 P098*** / ***P119*** P170 P206 P231	疲劳约束
5	P005 P042 P071 P105 P131 ***P180 P215*** / ***P240***	疲劳约束
13	***P013 P052*** / ***P081 P103*** P152 P182 P220 P242	里程/疲劳约束
16	P016 P043 P092 P127 ***P160 P178*** / ***P213***	疲劳约束
25	P025 P063 P111 P145 ***P179 P195*** / ***P232***	疲劳约束
35	P035 P087 P126 ***P177 P212*** / ***P236***	疲劳约束
37	P038 P073 P113 ***P158 P192*** / ***P227***	疲劳约束

截断位置在任务链的中间，那么截断后可以得到两条子任务链；如果截断位置在任务连端点处，那么阶段后得到一条子任务链和一个未安排的任务。

2. **拼接**：在将某个未安排的任务连接到已经安排好的任务链上时，趋向于使约束条件更易满足的方法。

以初步方案中的任务链 4，5，29 的处理方式为例。其中，任务链 4 和 5 中出现了不符合疲劳约束的子链，即连续工作累计两小时以上，且中间不包含任何小休。将这些发生冲突的子链从最后一个任务前截断，再根据截断处在整个链的位置，与符合题意的任务链 29 进行拼接操作。如图2所示。

处理之后，以增加一条任务链为代价，使方案满足了约束。如图3所示。

以上算法得到的最终方案由 41 条符合题意的任务链组成，按照题目要求，详细结果以表格文件的形式记录在了附件中。

5.4 算法合理性分析

如果允许 30 分钟以内的“加班”现象存在，那么任务链数量可以进一步缩减到 40 条。这已经很接近理论估计的下界 37 条，因此可以认为是较优的结果。

事实上，第二问用到的算法的可行性与合理性对附录 1 的数据存在较大的依赖性。这些数据自身带有的特性包括：

1. **对工作日约束的满足**：附件数据中只给了 6:03:11～14:53:32 约 9 个小时的车次数

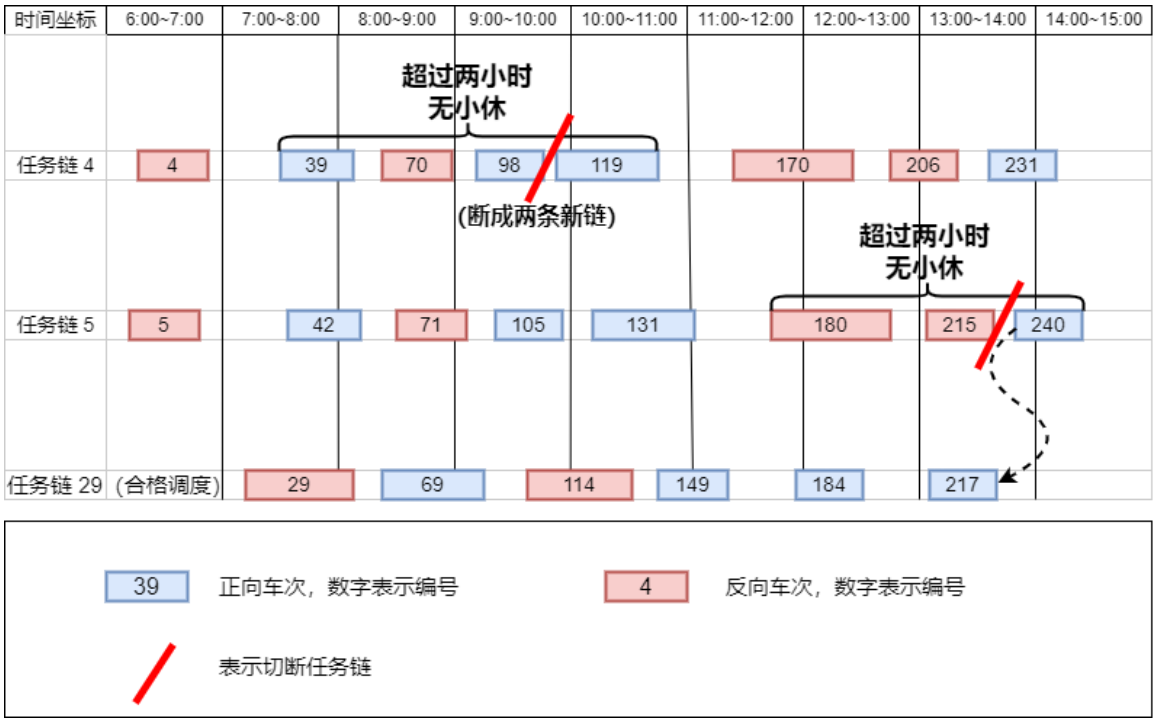


图 2　截断-拼接的具体做法

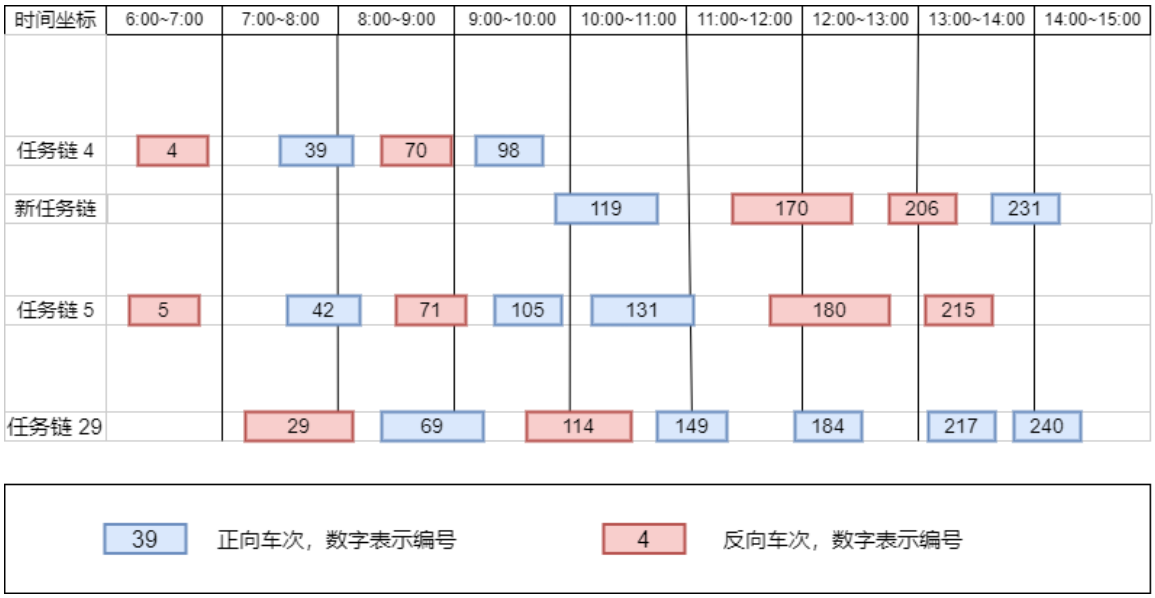


图 3　满足约束的处理结果

据，这使得工作日约束几乎不需要考虑。事实上，与工作日约束冲突的车次对并不多，在二分图4中列出了全部会发生冲突的车次，只需保证这些节点对不出现在同一条任务链中即可。

由图可知，会发生冲突的车次集中在当天日程的开始和末尾部分，因此对于不满足工作日约束的任务链，可以通过一次截断操作修正。

2. **对里程约束的满足**：根据统计，站点之间的运行时间如表2。

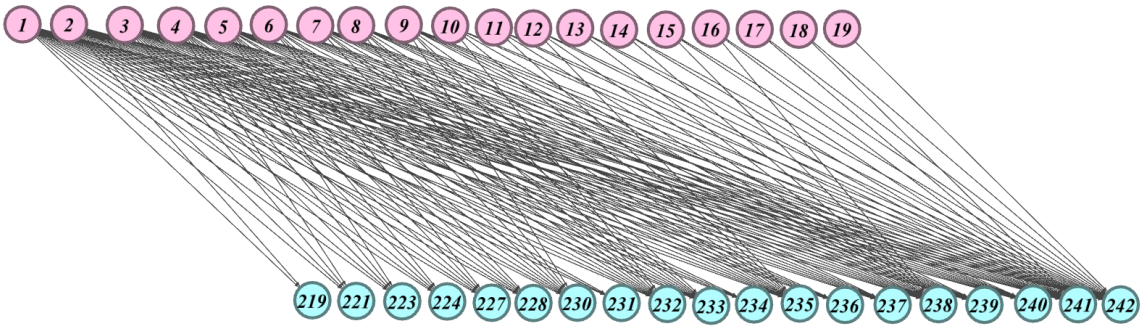


图 4　与工作日约束冲突的车次

表 2　运行数据

区间	运行时间 (min)	车次占比
$PA \to PB$	00:32:01∼00:35:01	57/242
$PB \to PA$	00:35:38	57/242
$PB \to PC$	00:54:32	68/242
$PC \to PB$	00:53:49∼00:57:49	60/242

对每次任务的平均耗时进行粗略的计算：使用加权求和的方法，得到平均运行时间为 45 分钟；根据运行时间的最小值 32 分钟，最多三次连续任务之后就要安排一次休息；由题意，最多两次连续任务之后就有一次反向换乘。

因此，每个任务的平均时间将会接近一个小时，这使得每位司机每个工作日内最多只能进行 8 次左右的任务，其中至少存在两次小休。再计算每个任务的平均里程为 25.63km。粗略估计的任务链最大长度并不容易取到，即使在每个任务链长度均为 8 的情况下，也有

$$25.63\ \text{km} \approx 25\ \text{km} = \frac{200\ \text{km}}{8}$$

这说明里程约束大概率也是自动满足的。

3. **对偏好约束的满足**：根据附件 1 给出的数据，计算前 k 个车次中，从 PA 发车的车次和 PB 发车的车次数量之比，得到结果如折线图5所示。

由折线图可以看出，出勤比呈从零开始的增长趋势，且前 30 次车的出勤比始终维持在很低的水平。出勤比始终不会超过题目给出的限制。

在设计算法的过程中，为了保证满足偏好约束，需要尽量让时间上靠前的车次作为任务链的起点。这符合 Dinic 算法按顺序遍历结点集合的特征。

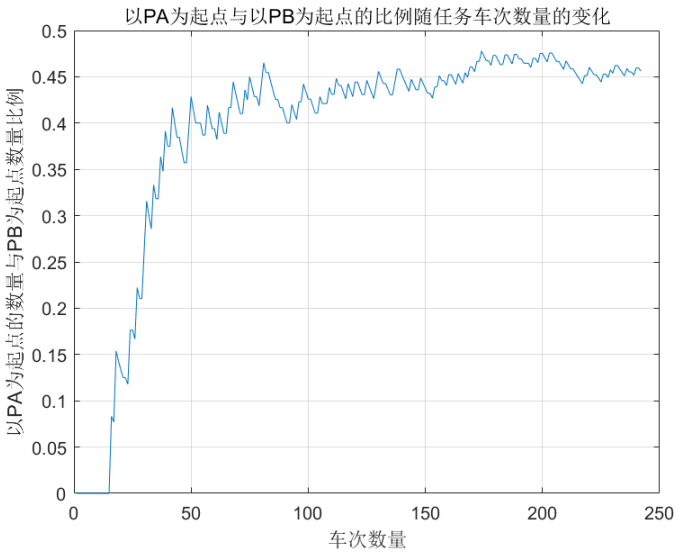


图 5　两站点的发车次数之比

5.5 问题三的求解

为了优化问题二中给出的算法在其他数据集上的表现，降低数据依赖性，需要更严格地对任务链是否满足约束进行判定。这里将疲劳约束、工作日约束、里程约束与偏好约束看作一个整体的判别条件，以后不再具体区分，只说一种调度**是否符合题意**。

5.5.1 调度方案合理性的评估指标

司机的排班方案调度可以看作一种任务分配问题，结合实际情况，提出针对调度方案的以下评价指标。

资源利用率。文献 [4] 在对 CPU 资源分配问题的研究中，指出了资源分配率这一指标的重要性。在本模型中，一条任务链就是一条流水线，认为司机在任务之间，除了必要的小休之外的空闲时间（包括反向换乘需要的步行时间）是被浪费的资源。优化的目标应当是减少这部分时间。

以每位司机的全天工作总时长为可利用资源量，计算第 2 问中得到方案的资源利用率。

休息时间公平性。针对资源分配公平性的评价指标，已经有一些成熟的研究。根据文献 [5] 中总结的公平性指标，为了衡量任务安排是否公平，应当计算各个司机工作量的差别。而在本题中，由于各个任务链司机所需的具体休息时间和换乘时间不同，司机内心的不公平感受不取决于总工作时长，而是超出必须的休息时间与换乘时间的额外休息时间。算法4给出了不必要空闲时间的计算方法。

在统计学中，一般使用变异系数 [6] 描述不平衡程度，其定义为标准差与平均值的比值：$c_v = \frac{\sigma}{\mu}$。

算法 4: 不必要空闲时长计算：CalculateTime

输入: 一条任务链 P

输出: 总不必要空闲时长 T

置 $T := 0$，取任务链中第一个任务对;
计算两个任务之间的相邻时长，加到 T 上;
if 此时必须休息 **then**
    $T := T - 1200$;
else
    if 该任务对为反向换乘 **then**
        $T := T - 600$;
if 任务链不空 **then**
    处理下一对任务对;
    跳转到 2;
return T;

当 $c_v > 1$ 时，认为分布是高差别的，否则认为是低差别的。经计算，得到各任务链休息时间的变异系数为 $c_v = 0.61 < 1$，可以认为安排较为公平。

时间安排合理性。一个合理的轮值安排应尽量使每次连续工作时长在一个稳定的范围内，不至于让司机有时过于疲劳，有时又过于空闲。为此，定义**任务均匀度**的概念

5.5.2 从算法角度提出的改进方案

在本问题中，我们从算法角度提出的改进方案主要是为了提高合理性、公平性和资源利用率，为此需要满足下面的要求：

1. 合理性：使司机在下一次休息之前的工作时间尽可能接近两个小时。

2. 公平性：使不同司机的额外休息时间尽量一致，即每次休息应该尽量接近 20 分钟同时不同司机的额外休息时间应该接近。

3. 资源利用率：使所有司机的总的除必要的小休外的休息时间之和尽可能小。

问题二的整体算法思路是包括两步：

1. 寻找数量尽可能少的满足部分约束的任务链。

2. 修正任务连使其符合剩余约束。

在问题三中我们仍然使用类似的的整体思路，如下所述：

1. 生成数量尽可能少的满足部分约束的任务链。

2. 在 1 中满足部分约束的所有任务链中，搜索满足剩余约束的任务链。

图6对比了两个问题的思路。

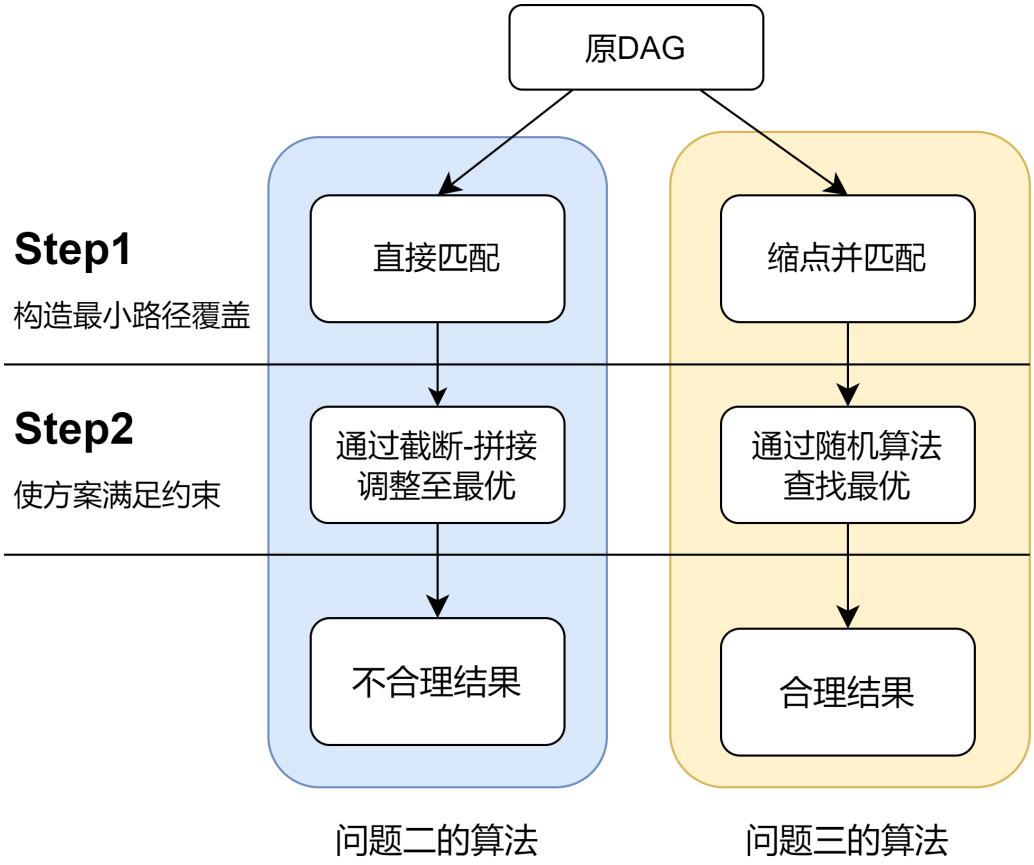


图 6　算法改进思路

为了提高合理性，考虑重构有向无环图，首先构造 $G^{(1)} = (V^{(1)}, E^{(1)})$，满足 $V^{(1)} = V$，而对于 v_i, v_j, 在符合连接约束的基础上，$(v_i, v_j) \in E^{(1)}$ 当且仅当 $c(v_i, v_j) = 0$，其中 $c(v_i, v_j)$ 表示边 (v_i, v_j) 的边权，即任务 i, j 的时间跨度小于 20 分钟时才连边。这意味着 $G^{(1)}$ 中的任意任务链中都不存在小休，可以求出 $G^{(1)}$ 的最小路径覆盖。

接着以 $G^{(1)}$ 及其最小路径覆盖，构造一个新的有向无环图 $G^{(2)}$，该构造方法基于以下两种基本操作：

1. **截断**。该操作与 5.3.2 中的截断方案是一致的，即使路径覆盖中的每一条路径满足疲劳约束，同时又尽量使每一段不休息的工作时间接近两小时，以满足合理性要求。截断操作后会得到若干长度较小的路径，根据第一问的最小不可行任务链的情况，这些长度较小的符合疲劳约束的路径的长度不超过 3.

2. **缩点**。将截断操作得到的长度较小的路径缩为一个整体，隐去内部的细节，称其为**超点**。每个超点具有和任务对应的顶点相似的属性，包括接车地点、接车时间、下车地点、下车时间、权重，这意味着可以用 5.1 中的描述来描述超点的属性，在不致混淆的前提下，超点以及超点构造的有向无环图具有 5.1 描述的所有属性与权重。

缩点之后，以所有的超点构造第二个有向无环图 $G^{(2)} = (V^{(2)}, E^{(2)})$，其中 $V^{(2)} =$

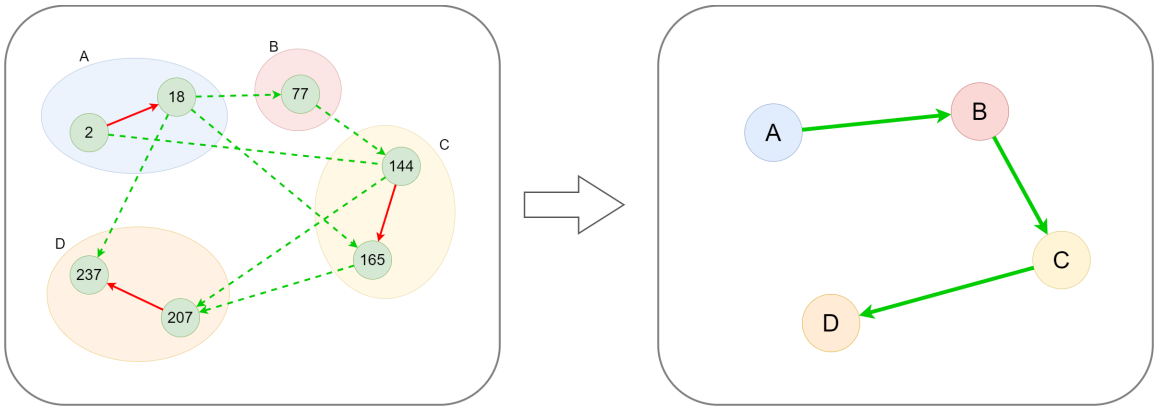


图 7　缩点处理的示例

$\{v'_1, \cdots, v'_{N'}\}$ 是所有的超点，其中 N' 是所有超点的数目，对于 v'_i, v'_j ，$(v'_i, v'_j) \in E^{(2)}$ 当且仅当 $c(v'_i, v'_j) = 1$，即每两个超点之间连边当且仅当两个超点之间满足连接约束和疲劳约束（需要有小休）。

图7给出了一个缩点后连边的的例子，其中在左边的图中，红色实线的有向边是时间小于 20 分钟的边，绿色虚线的有向边是时间不小于 20 分钟的边；右边的是缩点之后得到的超点之间的有向图，仅连时间超过 20 分钟的边。

得到 $G^{(2)}$ 之后，我们再求出 $G^{(2)}$ 的最小路径覆盖，再根据超点与原任务对应的点的对应关系将超点展开，便可以输出最终的符合原问题的路径覆盖方案，得到该方案的过程描述为算法5。

此时得到的路径覆盖中的路径一定满足连接约束和疲劳约束，但仍然有可能违反工作日约束和偏好约束，如果再采用第二问的截断-拼接方法不太合适，因为这种方法会使得路径的长度分布不均衡，同时不容易获得质量比较好的解。需要考虑使用其他的方法。

在经过若干次实验时，我们发现打乱生成的超点的顺序时得到的路径覆盖方案完全不同，对约束的违反情况也完全不同，而且在这个过程中出现了质量相当好（比如有的解仅有一条路径违反了工作日约束，且超时的时间仅有几分钟）的解，大部分的解对约束的违反都不严重，因此我们猜测在超点的所有排列对应的路径覆盖方案中，一定存在着不违反约束路径覆盖方案。

因此在众多的超点的排列中，考虑随机枚举，即便每次枚举得到的路径覆盖方案完全符合约束的概率非常小，但是由概率论可知在经过足够多的枚举之后，找不到完全符合约束的路径覆盖方案的概率趋于零，也就是说这种方法必定能找到完全符合约束的路径覆盖方案，这种随机算法属于 Las Vegas 算法的范畴，同时缩点构造新的有向无环图的操作显著缩小了搜索空间的大小，这将有助于我们快速找到完全符合约束的路径覆盖方案。事实证明这种方案确实能找到完全符合约束的路径覆盖方案。上述的随机搜索算法描述为算法6。

算法 5: 基于缩点的路径覆盖算法：Shirnk

输入: 图 $G = (V, E)$

输出: 路径覆盖方案 $\mathcal{P}$

根据前述规则构造 $G^{(1)}$，保证每条路径的时间跨度小于 20 分钟;

调用 `Dinic()` 算法求出 $G^{(1)}$ 的最小路径覆盖方案 $\mathcal{P}_1$;

根据疲劳约束对 $G^{(1)}$ 的 $\mathcal{P}_1$ 进行截断操作，获得若干较短的不违反疲劳约束的子路径;

以子路径进行缩点操作，获得超点集合 $V^{(2)}$;

调用 `random_shuffle()` 函数将 $V^{(2)}$ 随机打乱;

根据前述规则构造 $G^{(2)}$，保证每条路径的时间跨度不小于 20 分钟;

调用 `Dinic()` 算法求出 $G^{(2)}$ 的最小路径覆盖方案 $\mathcal{P}_2$;

将缩点展开，得到最终的路径覆盖方案 $\mathcal{P}_3$;

return $\mathcal{P}_3$;

算法 6: 随机搜索算法：LasVegas

输入: 图 $G = (V, E)$

输出: 不违任何约束的路径覆盖方案及其数目

初始化最大迭代次数 $M = 500$;

初始化最优路径覆盖方案的迭代轮数 $\texttt{best} = -1$;

for $1 \le i \le M$ **do**

    调用基于缩点的路径的覆盖算法，得到路径覆盖方案 $\mathcal{P} = \texttt{Shrink}(G)$;

    if $\mathcal{P}$ 不违反任何约束 **then**

        $\texttt{best} := i$;

        跳出循环;

return $\mathcal{P}$;

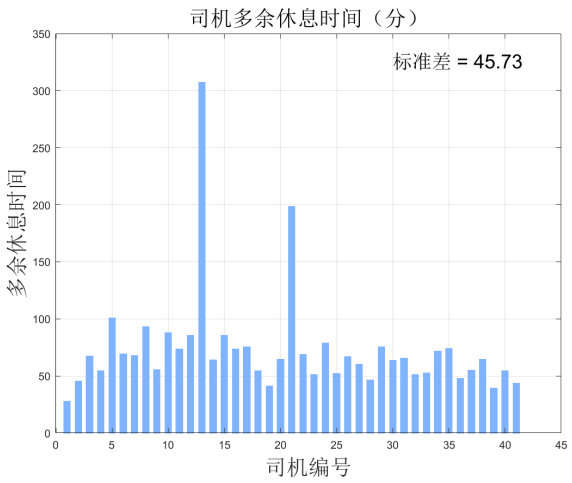
最终我们在第 437 次迭代时找到了不违反任何约束的方案 $\mathcal{P}$，该方案共有 42 条任务链，仅仅比 $C_{\min}$ 的下界 37 多了 5 条，比问题二的结果只多了 1 条，任务链中的任务数量集中在 5 或 6 个，是个相对较优的结果。

5.5.3 改进后的方案与问题二的结果对比

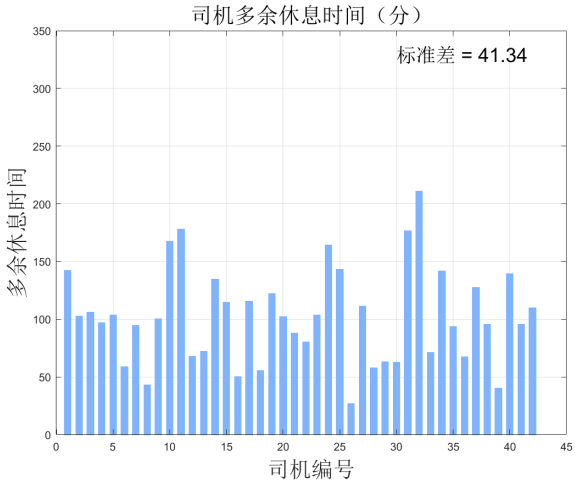
计算改进之后的模型的相关指标，得到表3与图8。

表 3　模型指标

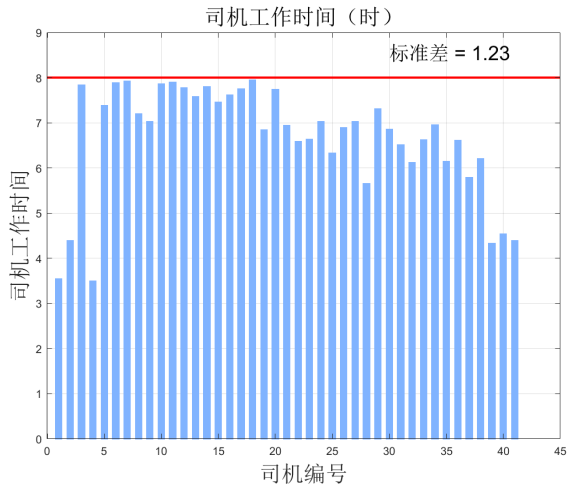
	原方案	改进方案
空闲时间变异系数	0.633	0.400
工作时长变异系数	0.185	0.140
任务均匀度	1916.850	1768.110



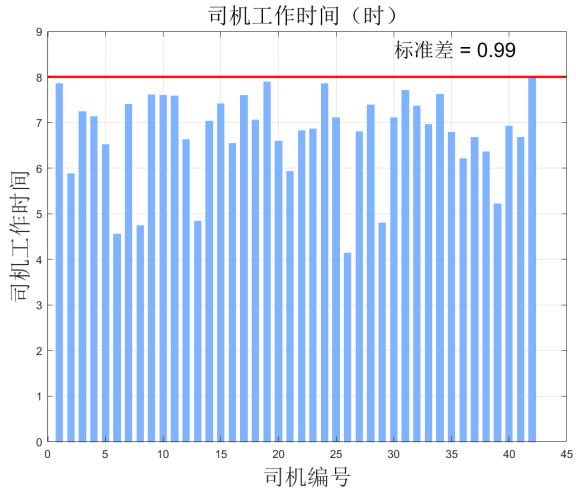
(a) 问题 2 各司机休息时间



(b) 问题 3 各司机休息时间



(c) 问题 2 各司机工作时间



(d) 问题 3 各司机工作时间

图 8

方差或变异系数越小，表示数据越稳定，就越符合我们的期望。根据这些指标，可以认为改进后的算法无论是合理性还是公平性都要优于原算法。

六、 模型评价与改进方向

6.1 模型的优点

1. 即使 MPC 问题转化得到的网络流问题可以很容易地使用线性规划求解，但是加了约束的问题二很难描述为标准的线性规划模型，因此采用了一种启发式规则，先产生少量违反约束的解，再对解进行修正，最终得到的解 41 与问题的解的下界 37 十分接近。

2. 求解有向无环图的最小路径覆盖使用的是复杂度最低的 Dinic 算法，算法的运行时间非常快。

3. 问题三的缩点算法和 Las Vegas 算法较为新颖，且在改进第二问的结果的基础上，总路径只增加了一条，仍然与下界 37 较为接近，同时即使循环了几百次，算法仍然在 5 秒内就得出了结果，表明这是一种有效算法。

6.2 模型的缺点

1. 先找到满足部分约束的解再修正对数据可能有一定程度的依赖性，对某些数据，这种解法可能会导致满足部分约束的解中违反剩余约束的数量较多，再修正时就会比较麻烦。

2. 对不同问题，缩点操作对搜索空间的减少程度可能有显著区别，随机搜索算法耗费的时间也可能会有显著区别，这会导致部分问题中随机算法表现会不好。

6.3 模型未来的改进方向

1. 通过改变顶点的顺序可以得到不同的路径覆盖方案，可以在问题二时就使用随机算法搜索。

2. 可以设计更多的启发式规则，使得找到的初步方案满足更多的约束，进一步地缩小搜索空间或者使路径覆盖方案的修正更加容易。

七、 参考文献

[1] Xiao F,Chen S,Han X, et al. A new direct acyclic graph task scheduling method for heterogeneous Multi-Core processors[J]. Computers and Electrical Engineering,2022,104(PA).

[2] Karp, R. M. Reducibility Among Combinatorial Problems[J]. Complexity of Computer Computations,1972:85-103.

[3] Alspach R B,Pullman J N. Path decompositions of digraphs[J]. Bulletin of the Australian Mathematical Society,1974,10(3).

[4] 鞠九滨, 杨鲲, 徐高潮. 使用资源利用率作为负载平衡系统的负载指标 [J]. 软件学报,1996(04):238-243.

[5] 赵丽花. 云计算环境下基于公平与效率的多资源调度优化研究 [D]. 华南理工大学,2021.1111.

[6] https://en.wikipedia.org/wiki/Coefficient_of_variation.

[7] 屈婉玲, 刘田, 张立昂, 王捍贫. 算法设计与分析 [M]. 第二版. 清华大学出版社, 2011(5) :176-184.

八、 附件清单

附件 1：定义问题一、二、三中公用的类和函数 publicLib.h

附件 2：利用拓扑排序和深度优先搜索求最小不可行子链程序 search_link.cpp

附件 3：利用网络流解决有向无环图的最小路径覆盖的程序 NetworkFlowSolver.h

附件 4：利用 Dinic 算法得出初步方案的程序 mpc.h

附件 5：利用截断-拼接方法改良问题二初步方案的程序 join_path.h

附件 6：检验任务链是否符合各种约束的程序 calculateIndex.h

附件 7：计算任务链工作时间、休息时间方差和均值 StandardDeviation.h

附件 8：最小不可行子链输出文件.txt

附件 9：问题二求解的轮值任务链输出文件.xlsx

附件 10：问题三求解的轮值任务链输出文件.xlsx

九、 附件

附件 1：定义问题一、二、三中公用的类和函数

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 300;

// 结束时间减开始时间
class _node
{
public:
    int trainNumber;
    int startTime;
    int startPlace;
    int endTime;
    int endPlace;
    int direction;
    int weight;
};
int nodeNum = 242; //车次的数量
_node task[maxn + 5]; //申请存车次信息的数组

// 二分图的边
typedef struct _bi_edge {
    int from, to;
    _bi_edge(int _from, int _to) :from(_from), to(_to) {}
}biEdge;

// 有向图的边
class diEdge {
public:
    int isEdged;
    int w;
};
diEdge digraph[maxn + 5][maxn + 5]; // DAG

// 生成有向无环图
void generate_dag() for (int i = 1; i <= nodeNum; i++)
{
    for (int j = 1; j <= nodeNum; j++)
    {
        if (j != i && task[i].endPlace == task[j].startPlace && task[j].startTime -
            task[i].endTime <= 3600) {
            if (task[j].startTime - task[i].endTime > 1200) {
                digraph[i][j].w = 1;
            }
```

```
            if (task[i].direction == task[j].direction && task[i].endTime < task[j].startTime) {
                digraph[i][j].isEdged = 1;
            }
            if (task[i].direction != task[j].direction && task[i].endTime + 600 <
                 task[j].startTime) {
                digraph[i][j].isEdged = 1;
            }
        }
    }
}

int deg[maxn + 5]; // 记录每个点的度数

// 计算入度
void compute_deg() {
    deg[0] = 9999999;
    for (int i = 1; i <= nodeNum; i++) {
        for (int j = 1; j <= nodeNum; j++) {
            deg[i] += digraph[j][i].isEdged;
        }
    }
}

//用于计算最小值
int min_index(int* arr, int n) {
    int tmp = 9999999, idx = 0;
    for (int i = 1; i <= n; i++) {
        if (tmp > arr[i]) {
            tmp = arr[i];
            idx = i;
        }
    }
    return idx;
}

//用于读取各个车次的信息
void read()
{
    freopen("E:\\CODES\\LOJ\\shanshu_cup\\output.txt","r",stdin);
    for(int i=1;i<=242;i++){
        int a,b,c,d,e,f;
        cin>>a>>b>>c>>d>>e>>f;
        task[i].trainNumber=a;
        task[i].startTime=b;
        task[i].startPlace=c;
        task[i].endTime=d;
        task[i].endPlace=e;
```

```
        task[i].direction=f;
        task[i].weight=task[i].endTime-task[i].startTime;
    }
    fclose(stdin);
}
```

附件 2：问题 1 利用拓扑排序和深度优先搜索求解最小不可行子链的程序

```
#include "publiceLib.h" // 引入自定义头文件 process.h

using namespace std; // 使用标准命名空间

queue<int> Q; // 声明一个整数队列 Q, 用于拓扑排序

void topology_sort() {
    int index = 1; // 初始化索引为 1
    for (int i = 1; i <= nodeNum; i++) { // 循环处理每个节点
        // index = *min_element(deg, deg + nodeNum); // 找到入度最小的点
        index = min_index(deg, nodeNum); // 调用自定义函数 min_index 寻找入度最小的点
        Q.push(index); // 将入度最小的点压入队列 Q
        deg[index] = 99999999; // 标记该点已经处理过, 将其入度设为一个较大的值
        for (int j = 1; j <= nodeNum; j++) {
            deg[j] -= digraph[index][j].isEdged; // 修改其他点的入度, 减去与该点相邻的边
        }
    }
}


int isv[maxn + 5]; // 用于标记节点是否已经访问过
vector<int> path; // 保存路径的向量

void dfs(int v, int totTime, int start, int &ans) { // 深度优先搜索函数
    if (totTime + (task[v].endTime - task[v].startTime) > 7200) { // 工作时间大于两小时
        if (totTime - (task[start].endTime - task[start].startTime) < 7200) { //
            如果去掉最左边的任务后总时间小于两小时
            ans++; // 增加不可行子链的计数
            for (auto x : path) cout << x << ' '; // 输出不可行子链
            cout << endl;
            return;
        } else {
            path.clear();
            return;
        }
    }
    totTime += (task[v].endTime - task[v].startTime); // 更新总时间
    for (int i = 1; i <= nodeNum; i++) { // 遍历所有节点
        if (!isv[i] && digraph[v][i].isEdged && task[i].startTime - task[v].endTime < 1200) {
```

```
            isv[i] = 1; // 标记节点已经访问
            path.push_back(i); // 将节点加入路径
            dfs(i, totTime, start, ans); // 递归访问下一个节点
            path.pop_back(); // 回溯，将节点从路径中移除
            isv[i] = 0; // 取消节点的访问标记
        }
    }
}

int search() { // 搜索最小不可行子链
    int ans = 0; // 初始化不可行子链的计数
    while (!Q.empty()) { // 当队列不为空时
        path.clear(); // 清空路径
        int totTime = 0; // 初始化总时间
        int u = Q.front(); // 获取队首节点
        memset(isv, 0, sizeof(isv)); // 重置节点访问标记数组
        path.push_back(u); // 将节点加入路径
        dfs(u, totTime, u, ans); // 调用深度优先搜索函数
        Q.pop(); // 弹出队首节点，继续下一个节点的处理
    }
    return ans; // 返回不可行子链的计数
}

int main() {
    freopen("E:\\CODES\\LOJ\\infeasible_link.txt", "w", stdout); // 打开输出文件
    read(); // 读取数据
    generate_dag(); // 生成有向无环图
    for (int i = 1; i <= nodeNum; i++) {
        for (int j = 1; j <= nodeNum; j++) {
            if (digraph[i][j].isEdged) {
                cout << i << ' ' << j << endl; // 输出有向边的信息
            }
        }
    }
    topology_sort(); // 进行拓扑排序
    cout << search(); // 输出搜索结果
    return 0;
}
```

附件 3：利用网络流解决有向无环图的最小路径覆盖的程序

```
#include "publicLib.h" // 引入外部处理文件

#define MX 20001
#define S 0
#define T ((n << 1) + 1)
#define oo 12312312
```

```
using namespace std;

typedef struct edge_t
{
    int u, v, c;
} edge;

edge e[MX];
int fst[MX], nxt[MX], lnum;
int n, m;

void addeg(int nu, int nv, int nc)
{
    nxt[++lnum] = fst[nu];
    fst[nu] = lnum;
    e[lnum] = (edge){nu, nv, nc};
}

void input()
{
    int a, b;
    scanf("%d%d", &n, &m);

    // 构建图的边和节点关系
    for (int i = 1; i <= m; i++)
    {
        scanf("%d%d", &a, &b);
        addeg(a, n + b, 1); // 添加正向边
        addeg(n + b, a, 0); // 添加反向边
    }

    // 添加源点到任务节点的边
    for (int i = 1; i <= n; i++)
        addeg(S, i, 1), addeg(i, S, 0);

    // 添加任务节点到汇点的边
    for (int i = n + 1; i <= n << 1; i++)
        addeg(i, T, 1), addeg(T, i, 0);
}

void init()
{
    memset(fst, 0xff, sizeof(fst));
    lnum = -1;
}
```

```
int dep[MX], q[MX];

int bfs(int frm, int to)
{
    int x, y, h = 0, t = 1;
    memset(dep, 0xff, sizeof(dep));
    q[++h] = frm;
    dep[frm] = 0;

    // 使用BFS查找增广路径
    while (h >= t)
    {
        x = q[t++];
        for (int i = fst[x]; i != -1; i = nxt[i])
        {
            y = e[i].v;
            if (e[i].c && dep[y] == -1)
            {
                dep[y] = dep[x] + 1;
                q[++h] = y;
            }
        }
    }
    return (dep[to] >= 0);
}

int dinic(int to, int x, int mn)
{
    if (x == to)
        return mn;
    int a, now = 0, y;
    for (int i = fst[x]; i != -1; i = nxt[i])
    {
        y = e[i].v;
        if (e[i].c && dep[y] == dep[x] + 1)
        {
            a = dinic(to, y, min(mn - now, e[i].c));
            now += a;
            e[i].c -= a;
            e[i ^ 1].c += a;
            if (now == mn)
                break;
        }
    }
    return now;
}
```

```
void output(int x)
{
    printf("%d ", x);
    for (int i = fst[x]; i != -1; i = nxt[i])
        if (e[i].c == 0 && e[i].v > n)
            output(e[i].v - n);
}

int fa[MX];

int findfa(int x) { return x == fa[x] ? x : fa[x] = findfa(fa[x]); }

void work()
{
    int tot = 0;
    while (bfs(S, T))
        tot += dinic(T, S, +oo);

    // 初始化并查集
    for (int i = 1; i <= n; i++)
        fa[i] = i;

    // 构建任务分配关系
    for (int i = 0; i <= lnum; i++)
        if (e[i].u >= 1 && e[i].u <= n && e[i].v > n && e[i].v < T && e[i].c == 0)
            fa[findfa(e[i].v - n)] = findfa(e[i].u);

    // 输出任务分配结果
    for (int i = 1; i <= n; i++)
        if (findfa(i) == i)
            output(i), putchar('\n');

    printf("%d\n", n - tot); // 输出未分配任务数
}
```

附件 4：利用 Dinic 算法得出初步任务方案的程序

```
#include<fstream>
#include"flow.h"
using namespace std;

void generate_dag1(int nodeNum){
// 生成不包含时间跨度大于20分钟的边的有向无环图，num为结点的数目
    freopen("E:\\CODES\\LOJ\\digraph_3.txt","w",stdout);
    int edgeNum=0;
    for (int i = 1; i <= nodeNum; i++)
    {
```

```
        for (int j = 1; j <= nodeNum; j++)
        {
            if (j != i && task[i].endPlace == task[j].startPlace && task[j].startTime -
                task[i].endTime < 20*60) {
                if (task[i].direction == task[j].direction && task[i].endTime <
                    task[j].startTime) {
                    digraph[i][j].isEdged = 1;
                    edgeNum++;
                }
                if (task[i].direction != task[j].direction && task[i].endTime + 600 <
                    task[j].startTime) {
                    digraph[i][j].isEdged = 1;
                    edgeNum++;
                }
            }
        }
    }
    cout<<nodeNum<<' '<<edgeNum<<endl;
    for (int i = 1; i <= nodeNum; i++)
    {
        for (int j = 1; j <= nodeNum; j++)
        {
            if(digraph[i][j].isEdged)cout<<i<<' '<<j<<endl;
        }
    }
    fclose(stdin);
    fclose(stdout);
}

void MPC(int n){ // n 为顶点数量
    freopen("E:\\CODES\\LOJ\\digraph_3.txt","r",stdin);
    freopen("E:\\CODES\\LOJ\\MPC_of_dag1.txt","w",stdout);
    init();
    input();
    work();
    fclose(stdin);
    fclose(stdout);

}

deque<int>path_of_dag1[80]; // dag1(20分钟以上的边不连)的路径覆盖
deque<int>small_path[300]; // 拆分后的零碎的路径

int path_of_len1[100]; // 长度为1的短路径
int path_of_len2[100][2];
//int path_of_len3[100][3];
deque<int>path_of_len3[100];
```

```
vector<int>final_path[100]; // 展开缩点之后得到的路径

int read_path(string p,int flag) // 读最小路径覆盖
{
       ifstream file(p); // 打开文本文件
   if (file.is_open()) {
       string line;
       int i = 0; // 用于索引vector中的元素
       while (getline(file, line)) { // 逐行读取文件
           //path_of_dag1[i].resize(10);
           istringstream iss(line);
           int value;
           while (iss >> value) { // 读取整数
               if(!flag){
                   path_of_dag1[i].push_back(value);
               }
               if(flag){
                   cout<<"It's okay here"<<i<<endl;
                   for(auto x:small_path[value-1]){
                       final_path[i].push_back(x);
                   }
               }
           }
           i++; // 移动到下一个vector
       }
       file.close(); // 关闭文件
       return i;
   }
   else {
       std::cerr << "无法打开文件" << std::endl;
   }

}


int cnt=0; // 记录总的零碎的段路径的数目

void break_up(){ // 将dag1的最小路径覆盖拆分，使其符合约束
   string p="E:\\CODES\\LOJ\\MPC_of_dag1.txt";
   read_path(p,0); // 读路径
   for(int i=0;i<69;i++){
       int tmp=0;
       while (!path_of_dag1[i].empty()){
           int x=path_of_dag1[i].front();
           small_path[cnt].push_back(x);
           tmp+=task[x].weight;
```

```
            if(tmp>2*60*60){
                small_path[cnt].pop_back();
                cnt++;
                tmp=0;
            }
            else{
                path_of_dag1[i].pop_front();
            }
        }
        cnt++;
    }
    freopen("E:\\CODES\\LOJ\\small_path_of_dag1.txt","w",stdout);
    for(int i=0;i<cnt;i++){
        for(auto x:small_path[i])cout<<x<<' ';
        cout<<'\n';
    }
    fclose(stdout);
}
int cnt1=0,cnt2=0,cnt3=0; // 长度为1,2,3的小路径的数目

void seperate_1_2_3(){
    for(int i=0;i<cnt;i++){
        if(small_path[i].size()==1){
//          path_of_len1[cnt1++]=small_path[i];
            path_of_len1[cnt1++]=small_path[i].front();
        }
        if(small_path[i].size()==2){
//          path_of_len2[cnt2++]=small_path[i];
            path_of_len2[cnt2][0]=small_path[i].front();
            path_of_len2[cnt2++][1]=small_path[i].back();
        }
        if(small_path[i].size()==3){
            path_of_len3[cnt3++]=small_path[i];
//          path_of_len3[cnt3][0]=small_path[i].front();
//          path_of_len2[cnt3][1]=small_path[i].back();
        }
    }
}

void add_back(){ // 长度为1的点加到长度为2的点的后面, 已经确认没有任何能加的
   for(int i=0;i<cnt1;i++){
      for(int j=0;j<cnt2;j++){
         if(task[path_of_len2[j].back()].endPlace==task[path_of_len1[i].front()].startPlace&&
         task[path_of_len2[j].front()].weight+task[path_of_len2[j].back()].weight+
         task[path_of_len1[i].front()].weight<2*60*60 &&
         digraph[path_of_len2[j].back()][path_of_len1[i].front()].isEdged==1
         ){
```

```
            path_of_len2[j].push_back(path_of_len1[i].front());
            path_of_len1[i].pop_back();
        }
    }
  }
  freopen("E:\\CODES\\LOJ\\len2_small_path_after_modified.txt","w",stdout);
  for(int i=0;i<cnt2;i++){
     for(auto x:path_of_len2[i]){
        cout<<x<<' ';
     }
     cout<<endl;
  }
  fclose(stdout);
  freopen("E:\\CODES\\LOJ\\len1_small_path_after_modified.txt","w",stdout);
  for(int i=0;i<cnt1;i++){
     for(auto x:path_of_len1[i]){
        cout<<x<<' ';
     }
     cout<<endl;
  }
  fclose(stdout);
  return ;
}

int isv[100]; // 判断某个长度为1的点是否已经被重新组合过了

void merge(){ // 判断长度为1的点之间能否互相组合，已经判断出所有孤立点都无法进行组合
   for(int i=0;i<cnt1;i++){
      for(int j=0;j<cnt1;j++){
         if(j!=i&&!isv[j]&&!isv[i]){
            if((task[path_of_len1[i]].endPlace==task[path_of_len1[j]].startPlace)&&
            (digraph[path_of_len1[i]][path_of_len1[j]].isEdged==1)){
               isv[i]=isv[j]=1;
               cout<<i<<' '<<j<<endl;
            }
         }
      }
   }
//   freopen("E:\\CODES\\LOJ\\len2_small_path_after_modified.txt","w",stdout);
//   for(int i=0;i<cnt2;i++){
//      for(auto x:path_of_len2[i]){
//         cout<<x<<' ';
//      }
//      cout<<endl;
//   }
//   fclose(stdout);
}
```

```
bool cmp(deque<int>a,deque<int>b){ // 对分裂后的零碎路径进行降序排序
    return a.front()<b.front();
}

typedef struct shrink_node{
    int st; // 开始时间
    int et; // 结束时间
    int sp; // 开始地点
    int ep; //结束地点
    int w; // 权重, 等于et-st
}ShrinkNode;

int node_num2=128;
ShrinkNode s_node[150]; // 缩点i(i>=1)对应small_path的i-1条

void shrink(){ // 缩点操作
//    sort(small_path,small_path+node_num2,cmp); // 128个短路径
//    random_device rd;
//    mt19937 gen(rd());
    srand(static_cast<unsigned int>(time(nullptr)));
    random_shuffle(small_path,small_path+node_num2);
    for(int i=1;i<=node_num2;i++){
        s_node[i].st=task[small_path[i-1].front()].startTime;
        s_node[i].et=task[small_path[i-1].back()].endTime;
        s_node[i].w=s_node[i].et-s_node[i].st;
        s_node[i].sp=task[small_path[i-1].front()].startPlace;
        s_node[i].ep=task[small_path[i-1].back()].endPlace;
    }
}

int digraph2[300][300];

void generate_dag2(){ // 生成缩点之后的点形成的有向无环图, 注意仅仅连20分钟以上的边
    int edge_num2=0;
    memset(digraph2,0,sizeof(digraph2));
    for(int i=1;i<=node_num2;i++){
        for(int j=1;j<=node_num2;j++){
            if(j!=i&&s_node[i].ep==s_node[j].sp&&s_node[j].st-s_node[i].et>=20*60){
                digraph2[i][j]=1; // 写入digraph_4文件中
                edge_num2++;
            }
        }
    }

    ofstream ofs1;
    ofs1.open("E:\\CODES\\LOJ\\digraph_4.txt", ios::trunc);
```

```
    ofs1<<node_num2<<' '<<edge_num2<<endl;
    for(int i=1;i<=node_num2;i++){
        for(int j=1;j<=node_num2;j++){
            if(digraph2[i][j]){
                ofs1<<i<<' '<<j<<endl;
            }
        }
    }
    ofs1.close();
}

void MPC2(){ // n 为顶点数量
    freopen("E:\\CODES\\LOJ\\digraph_4.txt","r",stdin);
    freopen("E:\\CODES\\LOJ\\MPC_of_shrink_node_dag.txt","w",stdout);
    init();
    input();
    work();
    fclose(stdin);
    fclose(stdout);

}

int unfold_node(){ // 将缩点展开，输出最终的路径覆盖
    string p="E:\\CODES\\LOJ\\MPC_of_shrink_node_dag.txt";
//    string p="E:\\CODES\\LOJ\\MPC_of_shrink_node_dag_out_of_order.txt";
    return read_path(p,1);
}
int best=0;
void monte_carlo(){
    int M=500; // 最大次数
    for(int i=0;i<M;i++){
        // 打乱操作出现在shrink函数中，只需要循环之后的函数即可
        for(int j=0;j<42;j++)
            final_path[j].clear(); // 清空
        rest.clear();
        shrink();
        generate_dag2();
        MPC2();
        unfold_node();

        // 修改文件读写
        ofstream ofs;

        ofs.open("E:\\CODES\\LOJ\\random_result.txt", ios::app);
        ofs<<"第"<<i+1<<"轮循环"<<endl;
        ofs<<"本次迭代的最终路径:"<<endl;
        for(int j=0;j<42;j++){
```

```
            for(auto x:final_path[j]){
                ofs<<x<<' ';
            }
            ofs<<endl;
        }
        ofs<<"本次循环安排的指标，包括约束违反情况和一些标准差:"<<endl;
        calculateIndex(ofs);
        ofs.close();

        // 修改文件读写
        if(!flag){
            best=i;
        }
    }

}

int main(){
    // int x=0;
    read();
    generate_dag1(nodeNum); // task此处传入全局变量，构建第一个
    MPC(nodeNum);
    break_up();
    monte_carlo();
    cout<<best;
    return 0;
}
```

附件 5：利用截断-拼接方法改良问题二初步方案的程序

```
//超过两个小时的不合格孤立点
//int isolated[10] = { 213, 227,232, 236, 240 }; // 孤立点
//int violated_path[10] = {7,19,28,38,40}; // 违反约束的路径
//总时间超过8小时的孤立点
int isolated[10] = { 230,238 }; // 孤立点
int violated_path[10] = { 1,8 }; // 违反约束的路径

void join_path() {
    for (int i = 0; i < 2; i++) {
        path[violated_path[i]-1].pop_back();
    }
    for (int i = 0; i < 2; i++) {
        int flag = 0;
        for (int j = 0; j < 40; j++) {
            if (flag)break;
            int tail = *(path[j].end() - 1);
            if (task[tail].endPlace ==
```

```
                task[isolated[i]].startPlace&&task[tail].endTime<task[isolated[i]].startTime) {
                if ((prefix[tail] + task[isolated[i]].endTime - task[isolated[i]].startTime < 2 *
                    60 * 60 || task[isolated[i]].startTime - task[tail].endTime>20 * 60)&&
                    task[isolated[i]].endTime-task[j].startTime<=8*60*60) {
                    path[j].push_back(isolated[i]);
                    cout << "将" << isolated[i] << "加入到了第" << j + 1 << "个路径上"<<endl;
                    flag = 1;
                }
            }
        }
    }
    for (int i = 0; i < 40; i++) {
        for (auto x : path[i]) {
            cout << x << ' ';
        }
        cout << '\n';
    }
}
```

附件 6：检验任务链是否符合各种约束的程序

```
int prefix[250]; // 前缀和，表示到该点(包括该点)的行驶总时间，注意碰到休息时间大于20分钟的要清空
vector<int>rest;
int flag;

void calculateIndex(ofstream &ofs)
{
    flag=0;
    int percentA = 0;
    int percentB = 0;
    int percentC = 0;
    for (int i = 0; i < 42; i++)
    {
        int moyu = 0;
        int total = 0;
        int Greater_2hour = 0;
        float totalRoad = 0;
        vector<int>::iterator it;
        for (it = final_path[i].begin(); it != final_path[i].end(); it++)
        {

            total += (task[*it].endTime - task[*it].startTime);

            //用于判断加上后一个车次后,是否行驶时间超过2小时
            if (Greater_2hour + (task[*it].endTime - task[*it].startTime) > 2 * 60 * 60)
            {
                ofs << "加上这一车次累计行使时间超过两个小时: ";
```

```
            ofs << "line:" << i + 1 << " " << *it << endl;
        }

        Greater_2hour += (task[*it].endTime - task[*it].startTime);
        prefix[*it] = Greater_2hour;

        // 判断是否超过两个小时，但未休息20分钟
        if (Greater_2hour >= 2 * 60 * 60)
        {
            if (it != final_path[i].end() - 1)
            {
                if (task[*(it + 1)].startTime - task[*(it)].endTime < 20 * 60)
                {
                    ofs << "超过两个小时，未休息20分钟: ";
                    ofs << "line:" << i + 1 << " from: " << *it << " to:" << *(it + 1) << endl;
                }
            }
        }

        //休息时间超过20min,劳累里程数归零
        if (it != final_path[i].end() - 1)
        {
            if (task[*(it + 1)].startTime - task[*(it)].endTime >= 20 * 60)
            {
                //cout << "line:" << i + 1 << " " << "累计行驶里程归零的点: " <<*it<< endl;
                moyu += (task[*(it + 1)].startTime - task[*(it)].endTime - 1200);
                Greater_2hour = 0;
            }
            else
            {
                if (it != final_path[i].end() - 1)
                    moyu += task[*(it + 1)].startTime - task[*(it)].endTime;
            }
        }

        //计算总里程
        if (it != final_path[i].end())
        {
            if (task[*it].startPlace == 0)
            {
                if (task[*it].endPlace == 1)
                {
                    totalRoad += 17.8;
                }
                else if (task[*it].endPlace == 2)
                {
                    totalRoad += 50.4;
```

```
            }
        }
        else if (task[*it].startPlace == 1)
        {
            if (task[*it].endPlace == 0)
            {
                totalRoad += 17.8;
            }
            else if (task[*it].endPlace == 2)
            {
                totalRoad += 32.6;
            }
        }
        else if (task[*it].startPlace == 2)
        {
            if (task[*it].endPlace == 0)
            {
                totalRoad += 50.4;
            }
            else if (task[*it].endPlace == 1)
            {
                totalRoad += 32.6;
            }
        }
    }

    //判断一个任务链是否总里程是否超过200
    if (totalRoad >= 200)
    {
        ofs << "总里程超过200公里: ";
        ofs << "第" << i + 1 << "链的里程" << totalRoad << endl;
        flag++;
    }

}

//判断一天的行驶总时间是否超过8小时
if (total > 8 * 60 * 60)
{
    cout << "line:" << i + 1 << " from: " << *it << " to:" << *(it + 1) << endl;
    cout << "false2" << endl;
}

//计算每个任务链的开始路径比例
it = final_path[i].begin();
if (task[*it].startPlace == 0)
{
```

```
            percentA++;
        }
        else if (task[*it].startPlace == 1)
        {
            percentB++;
        }
        else
        {
            percentC++;
        }

        rest.push_back(moyu);

        if (task[final_path[i].back()].endTime - task[final_path[i].front()].startTime >= 8 * 60
            * 60)
        {
            ofs << "line:" << i + 1 << " 工作总时间: " << (task[final_path[i].back()].endTime -
                task[final_path[i].front()].startTime) << "," << endl;
            flag++;
        }
        // 加上违反8h总时间约束的任务链数
        //cout << task[final_path[i].back()].endTime - task[final_path[i].front()].startTime <<
            ",";
    }

    ofs << endl<<endl;
    ofs<<"每个任务连的总的休息时间:"<<endl;
    for (vector<int>::iterator it = rest.begin(); it != rest.end(); it++) //
        计算每一条任务链休息的总时间
    {
        ofs<< *it << ' ';
    }
    ofs<<endl;
    cout << "percentA:" << percentA / 42.0 <<" ";
    cout << "percentB:" << percentB / 42.0 << " ";
    cout << "percentC:" << percentC / 42.0 << endl;

}
```

附件 7：计算任务链工作时间、休息时间的方差和均值的程序

```
//计算方差和均值
double StandardDeviation(const vector<double>& data) {
    // 计算平均值
    double sum = accumulate(data.begin(), data.end(), 0.0);
    double mean = sum / data.size();
```

```
    // 计算差的平方和
    double sumOfSquaredDifferences = 0.0;
    for (const double& value : data) {
        double difference = value - mean;
        sumOfSquaredDifferences += difference * difference;
    }

    // 计算方差
    double variance = sumOfSquaredDifferences / data.size();

    // 计算标准差
    double standardDeviation = sqrt(variance);

    return standardDeviation;
}
```