基于整数规划的巡检路径与选址问题求解

摘要

本文对医院巡检员巡检路径的规划和巡检点的选址问题进行研究。首先将地图转换为无向图，并用 Dijkstra 算法求解最短路，对模型进行预处理。针对前两问巡检路线的规划问题，基于旅行商问题模型进行变形，建立整数规划模型。通过 Branch and cut 算法对 NP-hard 的模型进行求解，实现求解的快速和准确。针对第三问常驻点的选址问题，基于分枝定界法，对整数规划模型用剪枝的方法进行求解，优化求解过程。第四问基于两种情景分别建立两种整数规划模型，并根据求解结果对两种模型进行评价。

针对问题一，本文对经典的旅行商问题模型进行了变形，建立了整数规划模型，以求解多人同时巡检的最优路线方案。由于模型是 NP-hard 问题，因此我们采用 Branch and cut 算法对模型进行求解，实现了快速准确的求解。最终求解得到两名巡检员的最优巡检路线为：巡检员 1 巡检 5-7-21-20-8-10-9-19-16-17-18-1-2-5，巡检员 2 巡检 5-6-15-14-13-12-11-4-3-5。最短巡检时间为 201.24 秒。

针对问题二，本文根据巡检前和巡检中的两种情况建立了两种整数规划模型。针对巡检前的路径规划，基于问题一的模型，在目标函数中加入了发现问题及处理的时间期望，通过 Branch and cut 算法求解得到两名巡检员的最优巡检路线为：巡检员 1 巡检 5-7-21-20-8-18-17-16-19-9-10-5，巡检员 2 巡检 5-2-1-15-14-13-12-11-4-3-6-5。最短巡检时间为 319.05 秒。针对巡检中的情景，我们建立了在实际发现问题后对巡检路线进行及时调整的模型，使得巡检路线在实际情况中得到优化，并列举了两个具体实例进行求解。

针对问题三，本文在问题一模型的基础上，根据各巡检点的空间分布情况，按顺序进行迭代求解，通过剪枝方法对对求解过程进行优化，使得求解效率获得较大提升。最终求解得到将常驻点设在 1 号巡检点处可使两位巡检员巡检花费时间最短，最优巡检路线为：巡检员 1 巡检 1-18-17-16-19-9-10-8-20-21-7-5-1，巡检员 2 巡检 1-2-6-4-3-11-12-13-14-15-1。最短巡检时间为 192.30 秒。

针对问题四，本文基于对“加大巡检力度”的两种理解，建立了两种模型。模型一假设每次巡检都需要遍历所有巡检点，通过分析，对问题一中模型的最优路线方案进行了复用和整合，得到三次巡检的最短巡检时间之和为 603.72 秒。模型二假设每次巡检不需要遍历所有巡检点，对每个巡检员在每个时间段的两条巡检路线进行规划，重新建立模型。由于决策变量数量过大，我们采用了前 8 个巡检点进行小规模的规划实验，与模型一进行对比和评估。最终结果仅比模型一减少了 5.3 秒，而相比来说模型一更加符合实际情况。

关键词：整数规划 旅行商问题 Branch and cut 算法 剪枝法 路径规划问题

一、问题重述

1.1 问题背景

巡检是现实中一项常见并且重要的工作。在医院的治理中，巡检对维护医院设备的正常运作，防止和减少隐患和事故的发生，保障患者的诊疗环境和人员的人身安全具有重要意义。但是，巡检工作也面临着难题，包括如何合理地为巡检人员分配巡检任务、对巡检的选址进行优化，以满足医院的巡检要求，并且使得巡检完成时间等代价最小。

巡检的背后是运筹学界研究了半个多世纪的著名的组合优化问题——旅行商问题（TSP, Traveling Salesman Problem）。本问题涉及基本的旅行商问题以及其变形。旅行商问题在各个领域有非常广泛的应用，例如交通、物流、航空管理、医院管理等。旅行商问题虽然是一个 NP-hard 问题，但是如今已经有很多精确算法与启发式方法对其进行解决。

现有某医院专门成立了综合巡检科，每天安排三名巡检员对医院的 21 个巡检定点进行巡检并打卡。巡检点进行了编号，巡检点之间有特定的巡检路径可供选择。其中大门处（定点 5）是巡检员日常的工作岗位，要求出巡从此常驻点出发，结束时回到此常驻点。任何时刻都至少有一名巡检员留在常驻点。

1.2 所需解决的问题

假设巡检人员的巡检速度为 1.5m/s，有以下四个问题：

问题一：对出巡的两名巡检员的巡检点分配方案和巡检路线进行规划，使得两人巡检一遍的时间最短。

问题二：如果在巡检过程中有一定的概率发现巡检点需要处理的问题，此时需要巡检员打电话联系处理人员前来处理，处理完毕后巡检员才能离开。每个巡检点发现问题的概率 p、处理人员的所在点、处理时间各有不同，具体数据如下图所示：

表 1 巡检点处理数据

巡检点	概率 p	处理时间	处理人员	巡检点	概率 p	处理时间	处理人员
1	0.03	5 分钟	2，19	14，15，18，20	0.1	4 分钟	2，19
2，11	0.01	3 分钟		6，8，9，12	0.04	3 分钟	6，16
3，4	0.04	3 分钟		16，17	0.02	10 分钟	
7，10，19	0.05	5 分钟		5，13，21	0.07	1 分钟	5，13

根据以上数据，对两名巡检员的出巡方案重新进行调整。

问题三：如果巡检员的常驻点可以不为定点 5（大门），而是设置一个巡检办公室作为常驻点，选出巡检办公室的最佳常驻地点。

问题四：如果要求一天要进行早中晚三次巡检，并且每个巡检都被不同的巡检员打卡一次，对巡检员的出巡方案重新进行调整。

二、问题分析

2.1 问题一的分析

问题一是一个典型的旅行商问题，希望规划出一个最优巡检路径方案，使得两个巡检员走遍所有巡检点所用时间最短。其目标是使得巡检一遍所用时间最小化。由于两个人同时出发开始巡检，因此巡检所用时间应当为两个巡检员巡检时间中较大的时间。由于每两个巡检点之间的最短路径是确定的，并且巡检速度固定，因此巡检所用时间仅取决于巡检路线。巡检线路则可以表示为一个集合，集合中每个元素表示是否要从一个巡检点走向另一个巡检点，只需要取 0 或 1 两个整数即可。由于有两个巡检员，结果需要对应两个集合。

因此，考虑建立整数规划模型，构造 0-1 决策变量。目标函数可表达为：所用时间较长的巡检员线路上，每两点间所需时间与对应决策变量的乘积之和。用约束条件限制巡检员的巡检路线，包括：均需从起点出发、遍历所有巡检点后再返回起点、路径中除了包含起点的环外没有其他环。通过求解旅行商问题的常用的算法求解即可得到两个巡检员的巡检路线，即可得到巡检点的分配方案。考虑通过 Python 调用求解器——Gurobi 求解此整数规划模型。

2.2 问题二的分析

问题二在问题一的基础上增加了发现问题的概率。发现问题后需要联系对应的处理人员来到现场，并进行一定时间的处理。因此目标函数的巡检时间需要在问题一的基础上增加处理人员前往巡检点所需时间和处理时间。由于发现问题是随机事件，在巡检前是否发生是未知的，因此对每一个确定的巡检路线方案，每次实际巡检所需时间都是随机的。我们以计算处理人员前往和处理时间的期望，作为最优巡检方案的衡量标准。

由于其他要求不变，因此除了在目标函数中增加期望节点处理时间外，约束条件较问题一不发生改变。考虑用同样的算法进行求解，此时求出的最优解并不是绝对的最优解，而是使得期望巡检时间最小的巡检方案。

在实际巡检过程中，可能会由于问题的真实发生使得巡检方案不再是最优。因此我们基于巡检中的真实情况建立第二种模型。在问题真实发生后，重新规划路线，使得两人的巡检时间较短。对于故障真实发生后的路径规划问题，目标函数未发生改变，但需要约束两人的路径从各自的当前位置、遍历当前剩下未巡检的所有点后、再返回起点。

2.3 问题三的分析

问题三是一个选址-路径问题（LRP, Location Routing Problem），需要规划一个最佳的常驻地点，以获得一个和原来固定起点相比更优的巡检方案，使得巡检时间最短。

问题一已经建立了寻找某个固定巡检点作为常驻地的巡检时间最短方案的模型。本题模型则需要在 21 个巡检点中找出一个巡检点，当其作为常驻点时能够找到一个最优巡检方案使得巡检时间在所有方案中最短。但如果仅仅简单遍历 21 个巡检点进行求解，会耗费较多时间。由于求解旅行商问题常见算法中包括基于分支定界法的 Branch and cut 算法，因此很容易可以在分支的过程进行剪枝，所以问题三考虑通过剪枝方法对大于已知巡检点选址中的最优解的情况进行排除，对求解过程进行优化。

2.4 问题四的分析

问题四在增加了巡检次数，要求一天要进行三次巡检，并且每个巡检点都需要被不同的巡检员打卡一次。根据题意，我们对本题分为两种情况讨论。

第一种情况是假设每次巡检都需要对 21 个巡检点全部打卡。由于至少有一名巡检员留在常驻点，而两个巡检员同时巡检一遍的时间小于一个巡检员巡检一遍，因此每次巡检一定有两个不同的巡检员遍历所有巡检点，等同于问题一的情况。同时，要满足每个巡检员在三次巡检过后经过所有巡检点，则每个巡检员应出巡两次，并且两次巡检路线要遍历所有巡检点。在这种情况下，无论是每次巡检的两条路线还是每个巡检员的两次出巡路线，都应当是遍历一次巡检点的最优方案，即问题一的巡检方案。此时只需要根据问题一的方案对三名巡检员进行分配即可。

第二种情况是假设在三次巡检中，只需要保证每个巡检点都被不同的巡检员打卡一次，不需要每次巡检都遍历所有巡检点。但此时巡检员仍需要在三次巡检中出巡两次，并且两次出巡遍历所有巡检点。因此问题转换成对每个巡检员的两次巡检路线以及两次的出巡时段进行规划，使得三次巡检的总时间最短。同样可以建立整数规划模型，但此时要增加一个时段的决策变量，目标函数为三次巡检的时间和最小。

三、模型假设

假设 1 最优巡检方案总以巡检时间最短为衡量标准。

我们规定除第一问以外，其余小问对巡检路线的规划也都以巡检时间最短为目标，制定一个统一的衡量标准，便于模型的建立以及对结果的比较。

假设 2 在无故障的情况下忽略人员在巡检点的逗留时间。

在实际生活中，巡检人员在巡检点巡检往往需要进行打卡、检查等操作，由于这样的时间不是模型的考虑关键因素，故忽略此时间。

假设 3 巡检员在巡检中发现问题时，总会联系距离该点最近的巡检点的处理人员来处理。

由于题目中没有说明联系处理人员的具体规则，本题假设巡检员总是联系距离最近的处理人员，以减少等待时间，符合巡检时间最短这一目标，也简化了模型中目标函数的处理。

假设 4 每个巡检点的处理人员至少有两个。

每个巡检点发现问题时都有两个巡检点的处理人员可以联系。在**假设 2** 的基础上，我们假设每个巡检点的处理人员至少有两个，因此在两个巡检员同时发现问题并联系同一个巡检点的处理人员时，可以保证两边都得到及时处理，以简化巡检时间的计算。

假设 5 巡检人员在从一个巡检点到另一个巡检点时，选择最短路径。

四、符号说明

本文建立模型的过程中主要涉及以下符号，符号说明如下:

表 2 主要符号说明

符号	说明
x_{ijk}	01 变量，表示第 k 个巡检员是否从巡检点 i 前往巡检点 j。
x_{ijkl}	01 变量，表示第 k 个巡检员在第 l 时段是否从巡检点 i 前往巡检点 j。
c_{ij}	巡检点 i 走到巡检点 j 所需最短时间。
t_i	巡检点 i 发现问题后，处理人员从前往到处理完毕所需最短时间。
t_{i1}，t_{i2}	巡检点 i 的处理人员所在点到巡检点 i 所需时间。
s_i	巡检点 i 的处理时间。
p_i	巡检点 i 发现需要处理的问题的概率。
V	包含所有巡检点的集合。
K	包含所有巡检员的集合。
L	包含早中晚三个时间段的集合。

五、模型准备

5.1 处理地图数据

为了获得巡检点与巡检点之间的最短路径，首先将题目所给的地图转换成由点和边的集合构成的无向图：$G = (V, E)$。其中非空集合V是图G的顶点集，集合E为图G的边集，连接G中的两个顶点。

我们将地图中的巡检点和路径的交点都转换成图G的顶点，将巡检点、路径交点之间的蓝色路径转换成图G的边，同时测量每条边的距离，根据比例尺计算出实际距离作为每条边的权重。

为了得到精确的无向图，我们将地图导入 Autocad 进行描绘，并且在比例尺为 1cm 的精度上操作，保证测量的实际距离总误差不超过 0.001m。得到无向图如图 1 所示。将描绘出的无向图按照地图中所给比例尺的单位长度调整比例，即可得到各边的实际距离并将数据导出。

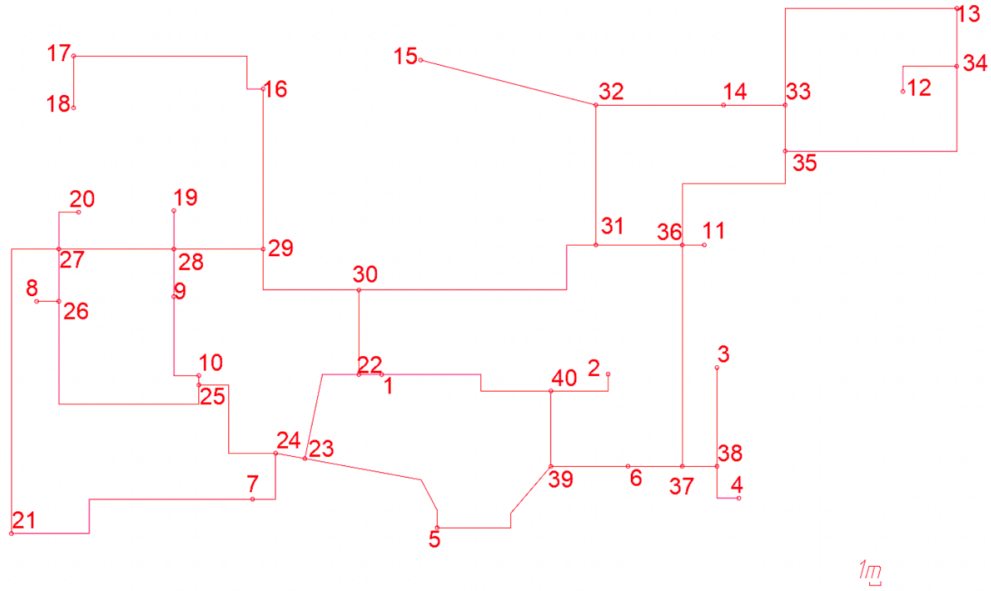

图 1 巡检点无向图

5.2 求解最短路径

由地图直接转换得到的图G中的顶点包括了巡检点和路径节点，并且不相邻的两点之间可以有多种路径选择。本题中只需要经过所有的巡检点，并且目标为巡检时间最短，因此结果中两巡检点之间的路径一定为最短路径。于是，我们可以对 5.1 中得到的赋权无向图G作进一步的处理，寻找图中每两个巡检点之间的最短路径。

寻找从一固定起点u_0到其余各点的最短路，最有效的算法之一是 E.W.Dijkstra 提出的 Dijkstra 算法。这个算法是一种贪心算法，通过求解子问题不断进行迭代，基于图中所有边的权重都是非负的假设，它的依据是一个重要而明显的性质：最短路上的任一子段也是最短路。[1]

算法的基本步骤为：

（1）设置一个顶点集合V_0，用于存放已求出最短路径的顶点。初始时仅包含起点u_0。

（2）从图G中除集合V_0外的所有顶点（即集合$\{V - V_0\}$）中选取从起点出发距离最短的顶点，加入集合V_0，并记录下起点到该点的最短距离。

（3）重复步骤（2），不断扩充集合V_0直至找到目标顶点v，则顶点u_0到顶点v的最短距离为最后一次求得的最短距离。

基于堆数据结构的单源 Dijkstra 算法的时间复杂度为$O(|V|\,log|V| + |E|)$，求解$|V|$个节点的最短路径速度快于其他算法。

我们采用 C++编写 Dijkstra 算法求解最短路（代码见附录一），最终整理得出 21 个巡检点之间的最短路径c_{ij}。

六、模型的建立与求解

6.1 问题一的建模与求解

6.1.1 模型分析

问题一要求规划出一个最优巡检路线，使得两个巡检员在打卡完所有巡检点并返回起点的所用时间最短。这是一个路径规划问题，考虑建立整数规划模型来进行求解。决策变量均为 0-1 变量，用来表示每个巡检员是否从一个巡检点走向另一个巡检点。目标函数为最小化两个巡检路线中的较大时间。约束条件保证两个巡检员从起点出发并最终回到起点、必须遍历 21 个巡检点、每个巡检点既是上一步的终点也是下一步的起点、路径中除了包含起点的环外没有其他环。

此模型的决策变量全部为 0-1 变量，通过求解该整数规划模型，将结果为 1 的决策变量涉及的巡检点按顺序连接，即可得到每个巡检员的巡检路线和巡检点分配方案。

6.1.2 模型建立

6.1.2.1 旅行商问题的基本模型

如果巡检员只有一个，则该问题为一个典型的旅行商问题（TSP）。旅行商问题的一个基本表达为：给定一系列城市和每两个城市之间的距离，求从一个城市出发，经过所有城市后回到起始城市的最短距离。在本题中，巡检点对应城市，由于巡检点之间的距离已知，并且巡检速度固定，则经过所有巡检点并返回常驻点的总时间最小与总距离最短是等价的。因此当巡检员只有一个时，可以转换成一个旅行商问题，可以用 0-1 整数规划来建立模型。

首先，设决策变量$x_{ij}=0$ 或 1，1 表示从点i走向下一个点j，0 表示不走。当$x_{ij}=1$时，总时间应当加上点i到点j之间所用时间c_{ij}，可以表示为$c_{ij}x_{ij}$；当$x_{ij}=0$时，不包括此段时间，但此时$c_{ij}x_{ij}=0$。

因此，设集合V表示所有巡检点的集合，则目标函数可表达为：

$$\min \sum_{i\in V}\sum_{j\in V} c_{ij}x_{ij}$$

约束条件包括：

（1）每个点恰好离开一次；

$$\sum_{j\in V} x_{ij} = 1,\quad i \in V$$

（2）节点平衡，即每个点进入与离开次数一致；

$$\sum_{j\in V} x_{ij} - \sum_{j\in V} x_{ji} = 0,\quad i \in V$$

（3）防止在遍历过程中出现子回路，即无法返回出发点的情形，附加一个强约束：

$$\sum_{i\in S}\sum_{j\in S} x_{ij} \le |S| - 1,\quad S \subsetneq V - \{0\}$$

如果只考虑前两个约束条件，该模型是一个整数线性规划，可以直接快速地进行求解。但是这样求解出来的结果可能出现多个封闭的子回路，如下图所示，不满足从起点出发，经过所有点后再返回起点。

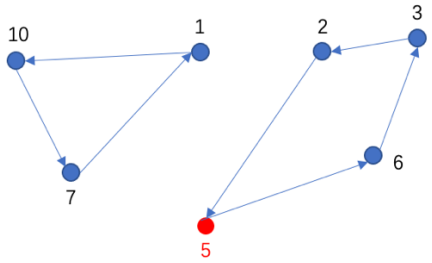

图 2 巡检路线中的子回路

因此需要增加约束（3）来消除可能出现的子回路。其中S表示任意一个除起始点外V中所有点的非空子集。当构成子回路时，该回路点的个数等于边的个数。因此，对所有的S，都保证其边的个数小于点的个数，即可消除子回路。其中，点的个数即为S中元素的数量，边的个数可表示为所有i和j都属于集合S的x_{ij}的和。但正因为有这条约束，旅行商问题变成了一个 NP-hard 问题。

6.1.2.2 问题一模型的建立

我们的模型以旅行商问题为原型，由于本题有两名巡检员，因此需要对模型进行进一步的改进。

设集合K表示巡检员的集合。将决策变量增加一个下标k，来区分第k个巡检员。决策变量可表示为：

$$x_{ijk} = \begin{cases} 1, & \text{巡检员}k\text{从巡检点}i\text{前往巡检点}j, \\ 0, & \text{否则.} \end{cases}$$

由于两个巡检员同时进行巡检，因此目标函数转换为两个巡检员巡检所用较大时间最小化。即最小化$\max \sum\sum c_{ij}x_{ijk}$，$k \in K$。

因此，目标函数可表示为：

$$\min d$$

其中，$d=\max\sum\sum c_{ij}x_{ijk}$，$k\in K$，表示两个巡检员中较大的巡检时间，该式可作为约束被线性化。

考虑两个巡检员后，约束条件包括：

（1）除起点外，每个点恰好离开一次；

$$\sum_{j\in V}\sum_{k\in K}x_{ijk}=1,\quad i\in V-\{0\}$$

（2）起点每个巡检员都离开一次；

$$\sum_{j\in V}x_{0jk}=1,\quad k\in K$$

（3）对每个巡检点，每个巡检员进入与离开次数一致；

$$\sum_{j\in V}x_{ijk}-\sum_{j\in V}x_{jik}=0,\quad i\in V,\quad k\in K$$

（5）对每个巡检员的巡检路线，都防止出现子回路；

$$\sum_{i\in S}\sum_{j\in S}x_{ijk}\le|S|-1,\quad S\subsetneq V-\{0\},\quad k\in K$$

综上所述，可建立本题的模型如下：

$$\min\ d$$

$$\begin{cases}d=\max\limits_{k\in K}\sum\limits_{i\in V}\sum\limits_{j\in V}c_{ij}x_{ijk}\\ \sum\limits_{j\in V}\sum\limits_{k\in K}x_{ijk}=1,\quad i\in V-\{0\}\\ \sum\limits_{j\in V}x_{0jk}=1,\quad k\in K\\ \sum\limits_{j\in V}x_{ijk}-\sum\limits_{j\in V}x_{jik}=0,\quad i\in V,\quad k\in K\\ \sum\limits_{i\in S}\sum\limits_{j\in S}x_{ijk}\le|S|-1,\quad S\subsetneq V-\{0\},\quad k\in K\\ x_{iik}=0,\quad i\in V,\quad k\in K\\ x_{ijk}=0\text{ 或 }1\\ V=\{0,1,2,\cdots,20\},K=\{0,1\}\end{cases}$$

此模型的求解结果保证了两人的巡检时间，也就是花费时间较长的巡检员所用时间尽可能短，但可能导致花费时间较短的巡检员所走路径并非最优。可以通过单独对花费时间较短的巡检员的巡检点集合求解单旅行商问题（见6.1.2.1）获取其的最优路径。本篇论文中关于路径的求解将采取这样的策略得到最终结果。

6.1.3 模型求解：Branch and cut 算法

分析所建立的模型可以发现，不考虑限制子回路的约束条件，此问题就是一个整数线性规划模型，可以比较容易地求解。但是子回路约束

$\sum_{i\in S}\sum_{j\in S}x_{ijk}\leq|S|-1,\ S\subseteq V-\{0\}$会随着$V$中点个数的增大导致集合$S$的规模呈指数级上升，并且在程序中不易表示。旅行商问题已被证明属于 NP-hard 问题，即无法在多项式时间内求得最优解。

目前已经有很多精确算法和启发式方法能够很好地处理子回路问题。其中，常用的精确算法有基于 Branch and bound（分枝定界）的 Branch and cut 算法、基于 Column Generation 的 Branch and Price 算法。考虑到该问题规模较小，可以得到精确解，故本题采用 Branch and cut 算法进行求解。

6.1.3.1 Branch and cut 算法

Branch and cut 算法实际上是以分枝定界法为基础的一种算法。分枝定界法的基本思想是对去掉部分约束条件后的松弛问题进行求解。将松弛问题不断分割为更小的问题，对子问题进行求解得到原问题解的上下限，并对确定不存在最优解的子问题（如最优解的下界超过了规定的上界）进行剪枝，直到所有子问题都不可再分，从而找到最优解。

Branch and cut 算法比分枝定界法多了增加切平面的步骤，剪去部分解空间，以此来收紧线性规划的松弛问题。通过求解收紧后的松弛问题来判断是否有剪枝的必要，增加剪枝概率，可以有效减少计算量。

6.1.3.2 深度优先搜索

Branch and cut 的求解过程中，利用深度优先搜索算法来判断求得的解中是否存在子回路。

深度优先搜索（depth-first search）算法的思想是对某一个图中的顶点进行正向标记，成为一个深度优先生成树，再任选一棵树深度搜索，并进行逆向标记，如果深度优先生成树都有回退边，则图中的路径形成了环，如果没有回退边，则不存在环。

6.1.3.3 求解步骤

本题通过 Python 调用 Gurobi 求解器进行求解。Gurobi 求解器作为行业顶尖的求解器，在求解整数规划方面有很好的效果。

首先将模型去除子回路约束，可以得到一个整数线性规划问题，此时解出的结果可能出现子回路，不一定满足题目要求。对该整数线性规划问题的松弛问题进行求解，可得到目标函数最优解为 140.449 秒，将其作为本问题的下限。接着对整数线性规划问题进行求解，得到最优解。

然后，利用深度优先搜索计算图中的子环。再将识别出的环成约束加入到模型中，开始新一轮迭代。

6.1.3.4 求解结果

根据以上求解算法，可以得到本题的最优解，包括巡检路线和巡检点分配方案，并算出巡检时间为 201.24 秒。具体数据如下表 3 所示：

表 3 问题一最优解

巡检员	巡检点分配	巡检路线	巡检时间
1	1, 2, 7, 8, 9, 10, 16, 17, 18, 19, 20, 21	5-7-21-20-8-10-9-19-16-17-18-1-2-5	201.24秒
2	3, 4, 6, 11, 12, 13, 14, 15	5-6-15-14-13-12-11-4-3-5	

对行走路径进行简化，仅表示巡检点之间的顺序，可得巡检路线如下图所示；

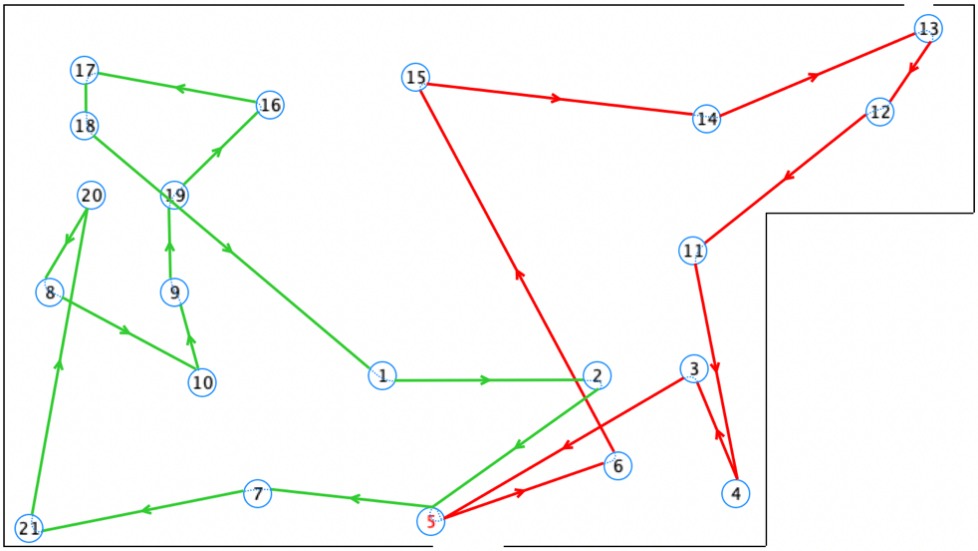

图 3 问题一巡检路线

6.2 问题二的建模与求解

6.2.1 模型分析

问题二的目标仍为巡检时间最小，但此时巡检时间还包括了在巡检过程中发现问题后的处理总时间，包括处理人员前往巡检点所需时间和处理时间。每个巡检点都对应两个处理人员所在巡检点，本题假设发现问题后总是联系距离该巡检点最近的处理人员（见**假设 3**）。因此每个巡检点发现问题后的处理总时间是固定的。但是，由于是否发现问题是一个随机事件，因此即使是在特定的巡检路线上，真实巡检时间也是随机的。

首先，为了在巡检前能够求得一个相对最优的巡检方案，我们在计算巡检时间时使用处理总时间的期望，以期望作为最优巡检方案的衡量标准。与问题一的模型相比，只有目标函数中的巡检时间发生了改变，决策变量仍然全部为 0-1 变量，约束条件不变。可以同样使用问题一的 Branch and cut 算法对问题进行求解，得到一个巡检前相对最优的巡检路线方案。

并且，为了使巡检路线方案更好地应对现实情况的随机性，我们在使用时间期望的模型解出一个相对最优巡检方案的基础上，又建立了一个在按最优巡检路线巡检时，问题真实发生后对路线进行重新规划的模型。在问题真实发生后，需要重新规划路线，以使得两人的巡检时间较短。对于故障真实发生后的路径规划问题，目标函数未发生改变，但需要约束两人的路径从各自的当前位置、遍历当前剩下未巡检的所有点后、再返回起点。对此模型进行求解，可以得出突发故障情况下的较优巡检方案。

6.2.2 巡检前最优路线规划模型

6.2.2.1 模型建立

目标函数为两个巡检员中较长的巡检时间最小化。巡检时间为巡检途中所用时间加处理总时间的期望。巡检途中的时间与问题一一致。根据**假设 3**，每个巡检点对应两处的处理人员，而巡检员总是联系距离该巡检点最近的处理人员，并且根据**假设 4**，每个巡检点的处理人员至少有两个，因此不会存在冲突等待的问题，因此每个巡检点处理总时间的期望等于该点发现问题的概率与最近的处理人员走到并处理完毕所需时间的乘积。具体可表示为：$t_i = p_i[\min(t_{i1}, t_{i2}) + s_i]$。因此，第 k 个巡检员巡检路线上的处理总时间为：$\sum t_j x_{ijk}$。

因此，目标函数可表示为：

$$\min d$$

其中，$d = \max\ (\sum\sum c_{ij}x_{ijk} + \sum\sum t_j x_{ijk})$，$k \in K$，表示两个巡检员中较大的巡检时间。

巡检规则没有发生改变，因此约束条件不变。

综上所述，模型建立如下：

$$\min d$$

$$\begin{cases} d = \max\limits_{k\in K} \sum\limits_{i\in V}\sum\limits_{j\in V}(c_{ij}x_{ijk} + t_j x_{ijk}) \\ \sum\limits_{j\in V}\sum\limits_{k\in K} x_{ijk} = 1, \quad i \in V - \{0\} \\ \sum\limits_{j\in V} x_{0jk} = 1, \quad k \in K \\ \sum\limits_{j\in V} x_{ijk} - \sum\limits_{j\in V} x_{jik} = 0, \quad i \in V, \quad k \in K \\ \sum\limits_{i\in S}\sum\limits_{j\in S} x_{ijk} \le |S| - 1, \quad S \subsetneq V - \{0\}, \quad k \in K \\ x_{iik} = 0, \quad i \in V, \quad k \in K \\ x_{ijk} = 0 \text{ 或 } 1 \\ V = \{0,1,2,\cdots,20\}, K = \{0,1\} \end{cases}$$

6.2.2.2 模型求解

此模型同样为整数规划模型，采用 Branch and cut 算法进行求解，可以得到本题的最优解，包括巡检路线和巡检点分配方案，并算出巡检时间为 319.05 秒。具体数据如下表 4 所示：

表 4 问题二巡检前最优路线规划模型结果

巡检员	巡检点分配	巡检路线	巡检时间
1	7, 8, 9, 10, 16, 17, 18, 19, 20, 21	5-7-21-20-8-18-17-16-19-9-10-5	319.05 秒
2	1, 2, 3, 4, 6, 11, 12, 13, 14, 15	5-2-1-15-14-13-12-11-4-3-6-5	

对行走路径进行简化，仅表示巡检点之间的顺序，可得巡检路线如下图所示：

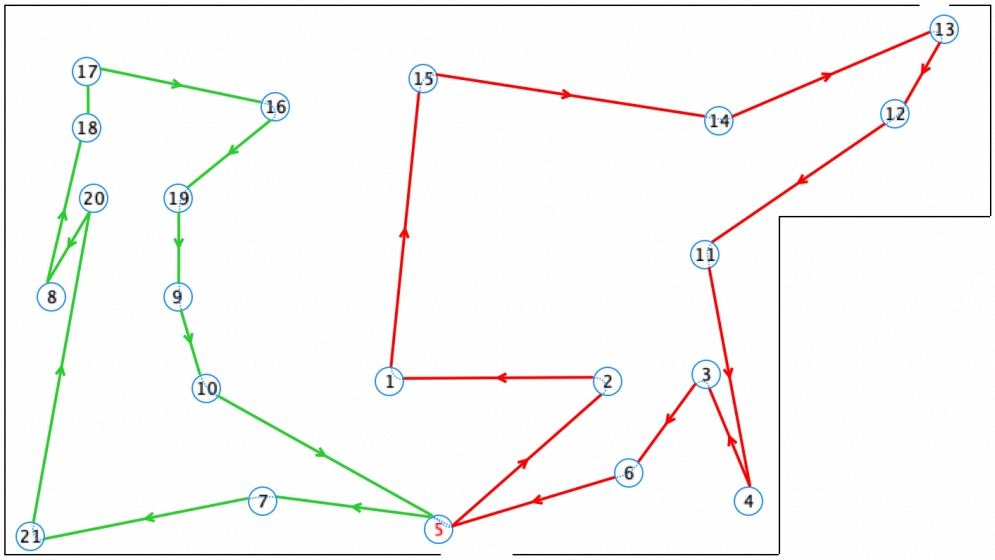

图 4 问题二巡检前最优巡检路线

6.2.3 巡检时重新规划路线模型

6.2.3.1 模型建立

6.2.2 中的模型得到了一个在巡检前使得巡检时间期望值最优的巡检路线。但是在按此巡检路线进行实际巡检时，发现问题是以概率p发生的随机事件，因此结果并不一定最优。本模型目的在于，在巡检时真实发现问题后，可以选择重新规划路线，以优化现有的最优巡检路线。

记一个巡检员在巡检点m处发现问题，此时另一巡检员正在巡检点n处。如果另一巡检员此时不在巡检点上，则按照巡检路线取到目前所在位置的下一巡检点。设 V_L是两个巡检员剩余未打卡的巡检点（包括巡检点m和n和常驻点5），此时根据两个巡检员的位置重新进行路径规划，使得走遍剩余巡检点所需时间最短。

因此，目标函数为最小化剩余巡检点所需巡检时间，同样应为最小化两个巡检员中所需较长时间。剩余巡检时间为剩余巡检点途中所用时间加上处理总时间的期望。对于在巡检点m的巡检员，巡检时间应再加上m点所需的处理时间。因此，目标函数可表示为：

$$\min \max_{k\in K} \sum_{j\in V_L} s_m x_{mjk} + \sum_{i\in V_L}\sum_{j\in V_L}(c_{ij}x_{ijk} + t_j x_{ijk})$$

约束条件包括：

（1）除起点外，剩余巡检点每个点都离开一次；

$$\sum_{j\in V_L}\sum_{k\in K} x_{ijk} = 1,\ \ i \in V_L - \{0\}$$

（2）除了起点和巡检点 m、n 外，对每个巡检点，每个巡检员进入与离开次数一致；巡检点 m、n 不能有巡检员进入，因为当前已到达 m、n；每个巡检员只能进入起点一次。

$$\sum_{j\in V_L} x_{ijk} - \sum_{j\in V_L} x_{jik} = 0,\ \ i \in V_L - \{m,n,0\},\ \ k \in K$$

$$\sum_{j\in\{m,n\}}\sum_{i\in V_L}\sum_{k\in K} x_{ijk} = 0$$

$$\sum_{i\in V_L} x_{i0k} = 1,\ \ k \in K$$

（3）由于此时不再从起点出发，因此对每个巡检员的巡检路线都需要防止出现回路；

$$\sum_{i\in S}\sum_{j\in S} x_{ijk} \le |S| - 1,\ \ S \subsetneq V_L,\ \ k \in K$$

综上所述，模型建立如下：

$$\min d$$

$$\begin{cases} d = \max\limits_{k\in K} \sum\limits_{j\in V_L} s_m x_{mjk} + \sum\limits_{i\in V_L}\sum\limits_{j\in V_L}(c_{ij}x_{ijk} + t_j x_{ijk}) \\ \sum\limits_{j\in V_L}\sum\limits_{k\in K} x_{ijk} = 1,\ \ i \in V_L - \{0\} \\ \sum\limits_{j\in V_L} x_{ijk} - \sum\limits_{j\in V_L} x_{jik} = 0,\ \ i \in V_L - \{m,n,0\},\ \ k \in K \\ \sum\limits_{j\in\{m,n\}}\sum\limits_{i\in V_L}\sum\limits_{k\in K} x_{ijk} = 0 \\ \sum\limits_{i\in V_L} x_{i0k} = 1,\ \ k \in K \\ \sum\limits_{i\in S}\sum\limits_{j\in S} x_{ijk} \le |S| - 1,\ \ S \subsetneq V_L,\ \ k \in K \\ x_{iik} = 0,\ \ i \in V_L,\ \ k \in K \\ x_{ijk} = 0 \text{ 或 } 1 \\ K = \{0,1\} \end{cases}$$

6.2.3.2 模型求解

此模型同样为整数规划模型，采用 Branch and cut 算法进行求解。两个巡检员都按照 6.2.2 模型求出的巡检前最优巡检路线进行巡检。但是每次巡检发现问题的巡检点都是随机的，因此本模型需要根据实际情况，在已知巡检点m和n的条件下进行求解。这里举两个具体的实例来对该模型进行求解。

实例 1 巡检员 1 在巡检点 16 发现问题，此时巡检员 2 巡检到点 15 处。

此时剩余巡检点$V_L = \{1,2,5,9,10,15,16\}$。代入模型求解可以得到剩余巡检点的最优路线，计算得到剩余巡检时间为 644.69 秒，具体结果如下表所示：

表 5 实例 1 求解结果

巡检员	巡检点分配	巡检路线	剩余巡检时间
1	16	16-5	644.69 秒
2	1, 2, 9, 10, 15	15-2-1-10-9-5	

对行走路径进行简化，仅表示巡检点之间的顺序，可得巡检路线如下图 5 所示：

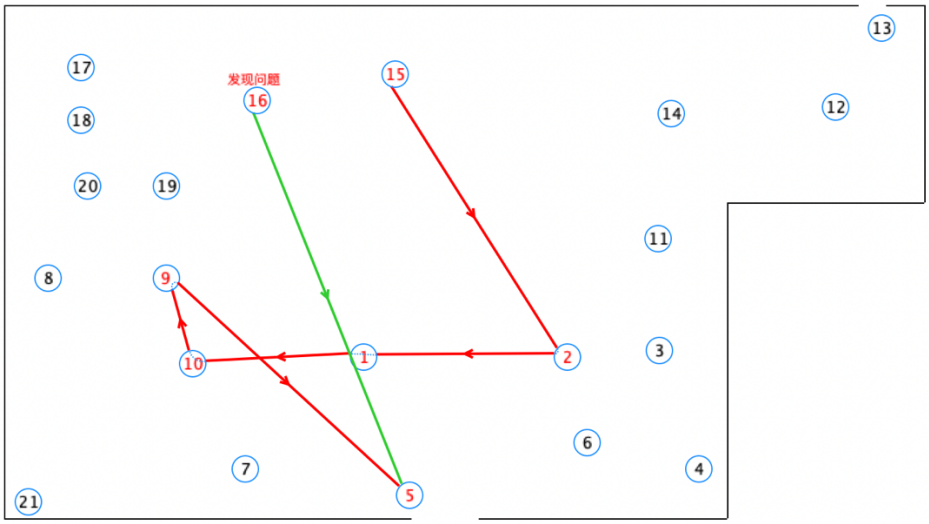

图 5 实例 1 重新规划巡检路线

实例 2 巡检员 1 在巡检点 8 发现问题，此时巡检员 2 巡检到点 11 处。

此时剩余巡检点$V_L = \{1,2,5,8,9,10,11,12,13,14,15,16,17,18,19,20\}$。代入模型求解可以得到剩余巡检点的最优路线，计算得到剩余巡检时间为 336.31 秒，具体结果如下表所示：

表 6 实例 2 求解结果

巡检员	巡检点分配	巡检路线	剩余巡检时间
1	2, 8, 9, 10, 19, 20	8-20-19-9-10-2-5	336.31 秒
2	1, 11, 12, 13, 14, 15, 16, 17, 18	11-12-13-14-15-17-18-16-1-5	

对行走路径进行简化，仅表示巡检点之间的顺序，可得巡检路线如下图 6 所示：

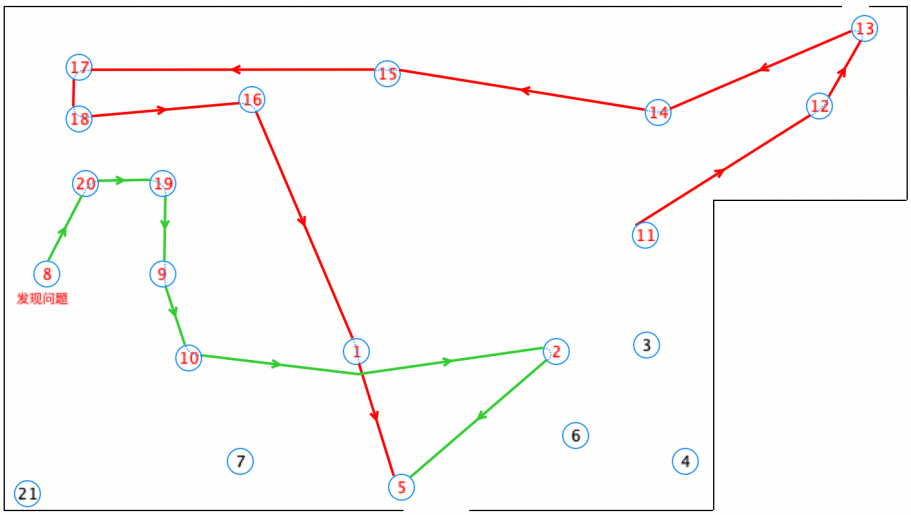

图 6 实例 2 重新规划巡检路线

6.3 问题三的建模与求解

6.3.1 模型分析

问题三的目标为在 21 个巡检点中找出一个巡检点，当其作为常驻点时的最优巡检方案所需巡检时间在所有方案中最短。对每个常驻点，其最优巡检方案的求解模型与第一问模型相同，因此约束条件与第一问一致，而目标函数变为基于常驻点的选择决策，最小化常驻点的最短巡检时间。

对模型的求解，如果只是遍历 21 个巡检点对每个常驻点进行求解，求解效率非常低下。因此问题三在对单个模型求解使用 Branch and cut 算法的基础上，基于 Branch and cut 背后的分枝定界法思想，通过剪枝方法对求解进行优化，更加高效地求出最佳常驻点。

6.3.2 模型建立

目标函数为最小化所有常驻点的最短巡检时间。设O表示任一常驻点。目标函数可表达如下：

$$\min_{O} \min_{x_{ijk}} d$$

其中，$d = \max_{k \in K} \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ijk}$，表示其中某一个常驻点的两条巡检路线中的较短巡检时间。$\min_{x_{ijk}} d$表示最小化该常驻点的较短巡检时间，也就是得到该常驻点的最短巡检时间。因此目标函数为最小化所有巡检点的最短巡检时间。

由于对单个常驻点求解最优方案的方法不变，因此模型约束条件也不变。

综上所述，第三问的模型建立如下：

$$\min_{O} \min_{x_{ijk}} d$$

$$\begin{cases} d = \max\limits_{k\in K}\sum\limits_{i\in V}\sum\limits_{j\in V} c_{ij}x_{ijk} \\ \sum\limits_{j\in V}\sum\limits_{k\in K} x_{ijk} = 1, \quad i \in V - \{O\} \\ \sum\limits_{j\in V} x_{0jk} = 1, \quad k \in K \\ \sum\limits_{j\in V} x_{ijk} - \sum\limits_{j\in V} x_{jik} = 0, \quad i \in V, \quad k \in K \\ \sum\limits_{i\in S}\sum\limits_{j\in S} x_{ijk} \le |S| - 1, \quad S \subsetneq V - \{O\}, \quad k \in K \\ x_{iik} = 0, \quad i \in V, \quad k \in K \\ x_{ijk} = 0 \text{ 或 } 1 \\ K = \{0,1\} \end{cases}$$

6.3.3 模型求解

对于约束条件$\sum_{i\in S}\sum_{j\in S} x_{ijk} \le |S| - 1$， $S \subsetneq V - \{O\}$， $k \in K$ ，仍然可以用 Branch and cut 算法进行求解。在第一问的求解中已经提到，Branch and cut 算法是在分枝定界法的基础上增加切平面。但是本题不再是对一个固定起点的最优巡检方案进行求解，而是要找出 21 个巡检点中的最优巡检方案，因此需要反复利用第一问的模型。

为了提高模型的求解效率，我们利用分枝定界法的思想，在其基础上增加剪枝，对本题的求解过程进行优化。回顾第一问中所提及的分枝定界法，对于一个最大化问题，如果子问题的最优解（松弛 LP 问题的解）小于下界，说明问题的最优解一定不在此子问题的解空间内，就要剪去。如下图所示，由子问题A_3求出了问题的一个下界 340，此时子问题A_4的最优解为 327，低于问题的下界，因此不必再对子问题A_4进行求解，将其剪去。

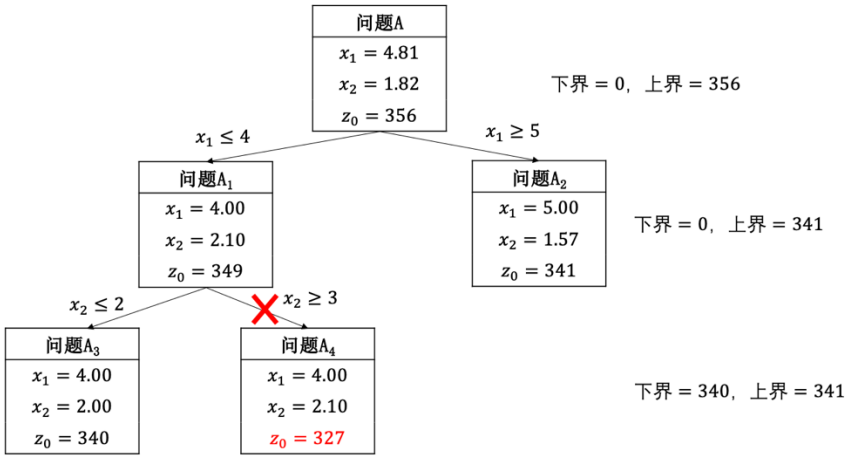

图 7 剪枝方法

在第一问的求解过程中，上界是由子问题求解结果得到，而在问题的模型中，我们规定上界为前 N 个常驻点求解的最值。求解的伪代码如下：

将第一问求解的d值赋值给d_{min}
将 21 个点按一定顺序依次赋值给O //一定顺序可以是根据各点距离图中心某点远近进行排序
  建立第一问模型
  增加约束：$d \leq d_{min}$
  Branch and cut 算法求解
  解得d
  如果$d < d_{min}$
    赋值给d_{min}
解得d_{min}以及对应的O点

通过 Python 调用 Gurobi 求解器进行实际求解，可以解得，第三小问的最佳常驻地点为巡检点 1，巡检时间为 192.30 秒，相比第一小问减少了 8.94 秒。具体结果如下表所示：

表 7 问题三最优解

巡检员	巡检点分配	巡检路线	巡检时间
1	5, 7, 8, 9, 10, 16, 17, 18, 19, 20, 21	1-18-17-16-19-9-10-8-20-21-7-5-1	192.30 秒
2	2, 3, 4, 6, 11, 12, 13, 14, 15	1-2-6-4-3-11-12-13-14-15-1	

对行走路径进行简化，仅表示巡检点之间的顺序，可得巡检路线如下图所示；

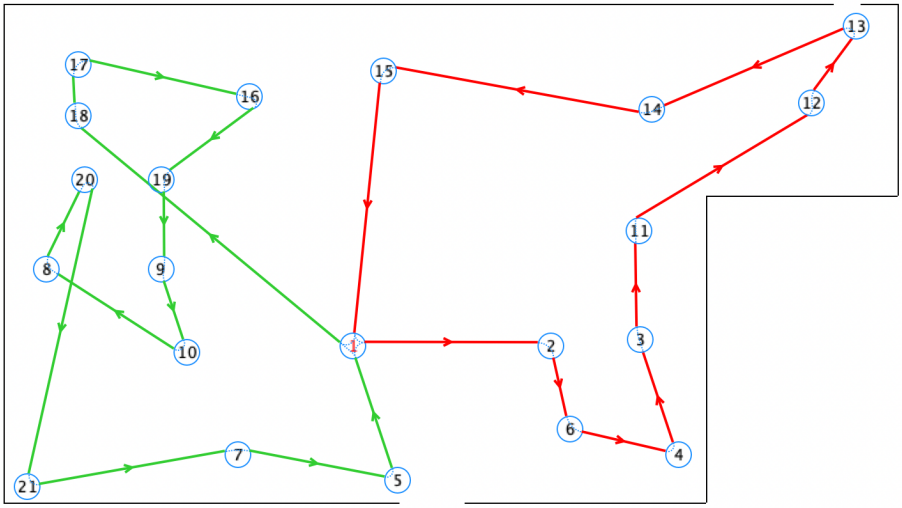

图 8 问题三巡检路线

6.4 问题四的建模与求解

本文基于对“加大巡检力度”的两种理解，对问题四建立了两种情景下的两种模型求解最优巡检方案，并根据求解结果对两种模型进行对比。

6.4.1 情景一：每次巡检需要打卡所有巡检点

6.4.1.1 模型分析

首先，由于至少有一名巡检员留在常驻点，而两个巡检员同时巡检一遍的时间小于一个巡检员巡检一遍，因此每次巡检一定有两个不同的巡检员遍历所有巡检点，等同于问题一的情况。

同时，要满足每个巡检员在三次巡检过后打卡所有巡检点，则每个巡检员应出巡两次，并且两次巡检路线要遍历所有巡检点。

在这种情况下，无论是每次巡检的两条路线，还是每个巡检员的两次出巡路线，都应当是从起点出发从两条路线遍历所有巡检点的最优方案，即为问题一的最优巡检方案。因此，可以延用问题一的模型，根据问题一解出的最优巡检方案，对三名巡检员在三个时段进行合理分配即可。

6.4.1.2 模型建立和求解

由分析可知，只需要继续使用问题一模型求解出来的最优方案中的两条巡检路线，保证每个巡检员在两个不同的时段采用这两条巡检路线即可。因此问题可以转变成一个简单的指派问题：三个巡检员都要完成两个任务，并且保证在三次内每个任务都要被每个巡检员完成一次。

结合第一问的最优巡检方案，可以得到情景一下本题的最优巡检路线，此时三次总巡检时间为 603.72 秒，每次巡检时间为 201.24 秒。具体结果如下：

表 8 问题四情景一求解结果

时间	巡检员	巡检点分配	巡检路线	巡检时间	
早	1	1,2,7,8,9,10,16,17,18,19,20,21	5-7-21-20-8-10-9-19-16-17-18-1-2-5	201.24 秒	603.72 秒
	2	3,4,6,11,12,13,14,15	5-6-15-14-13-12-11-4-3-5		
中	2	1,2,7,8,9,10,16,17,18,19,20,21	5-7-21-20-8-10-9-19-16-17-18-1-2-5	201.24 秒	
	3	3,4,6,11,12,13,14,15	5-6-15-14-13-12-11-4-3-5		
晚	3	1,2,7,8,9,10,16,17,18,19,20,21	5-7-21-20-8-10-9-19-16-17-18-1-2-5	201.24 秒	
	1	3,4,6,11,12,13,14,15	5-6-15-14-13-12-11-4-3-5		

6.4.2 情景二：每次巡检不需要打卡所有巡检点

6.4.2.1 模型分析

此时在三次巡检中，只需要保证每个巡检点都被不同的巡检员打卡一次，不需要每次巡检都遍历所有巡检点。此时，可以在一次巡检中两个巡检人员都对一个点进行打卡，也可以一次巡检中没有巡检员对该点打卡。但与情景一一样，巡检员仍需要在三次巡检中出巡两次，并且两次出巡遍历所有巡检点。因

此，在情景二下，应当是对每个巡检员的两次巡检路线以及两次的出巡时段进行规划，使得三次巡检的总时间最短。

同样可以建立整数规划模型，但此时要增加一个时段的决策变量，目标函数为三次巡检的时间和最小。

6.4.2.2 模型建立

为了区分时间段，需要对决策变量再增加一个下标l，此时决策变量可表示为x_{ijkl}，表示巡检员k在第l时段从巡检点i走向巡检点j。目标函数为最小化三次巡检所用时间之和。而每次的巡检时间应为两条巡检路线中所用较大时间。约束条件应保证每个巡检员在两次不同的时间段出巡两次，每次出巡每个巡检点的进入次数不多于 1 次。其余约束与模型一一致。

综上所述，情景二的模型建立如下：

$$\min \sum_{l\in L} \max_{k\in K} \sum_{i\in V}\sum_{j\in V} c_{ij}x_{ijkl}$$

$$\begin{cases} \sum_{j\in V}\sum_{l\in V} x_{ijk} \le 1, \quad i \in V-\{0\}, \quad k\in K \\ \sum_{j\in V}\sum_{k\in K}\sum_{l\in L} x_{ijkl} = |L|, \quad i\in V-\{0\} \\ \sum_{j\in V}\sum_{k\in K} x_{0jkl} = |K-1|, \quad l\in L \\ \sum_{j\in V} x_{ijkl} - \sum_{j\in V} x_{jikl} = 0, \quad i\in V, \quad k\in K \\ \sum_{i\in S}\sum_{j\in S} x_{ijkl} \le |S|-1, \quad S\subsetneq V-\{O\}, \quad k\in K, \quad i\in L \\ x_{iikl} = 0, \quad i\in V, \quad k\in K, \quad i\in L \\ x_{ijk} = 0 \text{ 或 } 1 \\ K=\{0,1,2\}, L=\{0,1,2\} \end{cases}$$

6.4.2.3 模型求解

此模型的决策变量x_{ijkl}共有$21*21*3*3$个, 对模型进行完整求解计算量过于庞大，因此我们只取前 8 个巡检点进行规划求解，来观察和评价用此模型求解对结果的优化效果。同样用 Branch and cut 算法，代入前 8 个巡检点对模型进行求解，可得在情景二下的巡检时间，具体结果如下：

表 9 问题四情景二前 8 个巡检点最优解

时间	巡检员	巡检点分配	巡检路线	巡检时间	
早	1	1, 7, 8	5-7-8-1-5	96.52 秒	267.31 秒
	3	1, 7, 8	5-1-8-7-5		
中	1	2, 3, 4, 6	5-6-2-3-4-5		

	2	1,7,8	5-1-8-7-5	96.52秒	
晚	2	2,3,4,6	5-2-6-3-4-5	74.27秒	
	3	2,3,4,6	5-1-8-7-5		

我们同样用情景一下的第一问模型代入前 8 个点进行求解，可以得到总时间为 272.61 秒。与第一问的最优路线结果进行对比，仅减少了 5.3 秒。通过小规模的实验，此模型能够进行优化，但是优化效果并不太好。采取情景一中的策略相对更加符合实际情况，求解也更加迅速。

七、模型的评价、改进与推广

7.1 模型评价

7.1.1 模型优点

（1）通过对经典的旅行商问题模型进行变形，较好地满足了该医院综合巡检科对于巡检的要求，并对巡检人员的路径、常驻点选址进行优化，尽可能减少了巡检人员花费的时间。
（2）通过 Branch and cut 算法求解 NP-hard 的旅行商问题，在保证结果准确的情况下实现了快速求解。
（3）通过剪枝方法对常驻点选址-路径问题进行求解，优化了求解过程，大大提高了求解效率，求得的结果为全局最优。
（4）问题二的模型二给出了实际情况中一般化的求解，在具体问题下代入数据可以得到该情况下的最优答案，面对不同问题展示出了灵活性。

7.1.2 模型缺点

（1）在问题二中，我们假设每个巡检点发现问题后都联系最近的处理人员，并且假设处理人员足够多，并没有考虑当两个巡检点同时发现问题时，处理人员不足发生的冲突问题。
（2）在问题三中，仅对模型求解过程进行了优化，但模型本身还是基于遍历，较为复杂。

7.2 模型推广

巡检是现实生活中的一项常见的工作。无论是社会的公共治安，还是单位的管理，都可能涉及巡检工作。

旅行商问题作为被运筹学界研究了半个世纪的问题，在实际生活中有很多应用场景：物流中心的送货、机场地勤的排班。除此之外，旅行商问题还有各

种各样的变形，例如带时间窗的旅行商问题、考虑随机性的旅行商问题、考虑鲁棒性的旅行商问题等。另外，文章中提到的选址-路径问题也被广泛应用于物流中心、应急中心等的选址上。

参考文献

[1] 司守奎, 孙玺菁. 数学建模算法与应用[M]. 国防工业出版社, 2021.

[2] 师文喜,付艳云,赵学义,刘海强,李鹏.基于多智能体的巡逻任务分配和路径优化方法研究[J].中国电子科学研究院学报,2021,16(07):633-638.

[3] Tian Liu, Zhixing Luo*, Hu Qin, Andrew Lim. A Branch-and-cut Algorithm for the Two-echelon Capacitated Vehicle Routing Problem with Grouping Constraints. European Journal of Operational Research, Volume 266, Issue 2, 16 April 2018, Pages 487-497.

[4] Cormen, Thomas H., Thomas H. Cormen. Introduction to Algorithms. Cambridge, Mass: MIT Press, 2001.

[5] Paolo Toth, Daniele Vigo. Vehicle Routing: Problems, Methods, and Applications, Second Edition, SIAM, 2014.

[6] Ismail Karaoglan, Fulya Altiparmak, Imdat Kara, Berna Dengiz. The location-routing problem with simultaneous pickup and delivery: Formulations and a heuristic approach. Omega. Volume 40, Issue 4, 2012, Pages 465-477

[7] Gurobi, https://www.gurobi.com/documentation/, 2021.

附录

支撑材料列表

（1）data/:论文所用到的数据，包括：
distance.csv（巡检点之间最短路径长度矩阵）、
distance（二进制文件，用于 Python 加载）、
expected_service_time.csv（巡检点发生故障后的处理时间）

（2）shortest_path/:求解巡检点之间最短路径的代码

（3）optimization/:模型求解所用的代码，包括：
preprocessing_distance（对距离数据的预处理）、
simple_vehicle_routing_problem（简单多旅行商问题代码）、
vehicle_routing_problem_with_service_time（第二问带期望处理时间模型代码）、
vehicle_routing_problem_with_failure_occurrence（第二问基于真实发生故障的路径再规划模型代码）、
location_routing_problem（第三问代码）、
vehicle_routing_problem_with_multi_time_periods（第四问代码）

（4）result/: 结果[1]输出，包括：
RESULT_P1（第一问）、
RESULT_P2_1（第二问带期望处理时间模型结果输出）、
RESULT_P2_2_1（第二问基于真实发生故障的路径再规划模型的算例 1 结果输出）、
RESULT_P2_2_2（第二问基于真实发生故障的路径再规划模型的算例 2 结果输出）、
RESULT_P3_1（第三问寻找最佳常驻点结果输出）、
RESULT_P3_2（第三问以 1 作为常驻点结果输出）、
RESULT_P4_1（第四问模型基于小规模巡检点的结果输出）、
RESULT_P4_2（第一问模型基于小规模巡检点的结果输出）、
SHORTEST_PATHS（最短路径求解结果输出）

附录 A 求解顶点之间最短路径-C++源代码

```
vector<pair<int, double>> Graph::DijkstraShortestPath(int s)
{
   // copied from Prim's minimum spanning tree algorithm
```

[1] 此处结果并未对时间花费较短的巡检人员进行二次 TSP 求解，而是原始模型的之间输出。

```
    vector<tuple<double, int, bool>> nodeVec = vector<tuple<double, int,
bool>>(graph.size(), tuple<double, int, bool>(inf, null, true)); //
tuple<key, π, isInQ>

    // initialize the nodeVec and the priority queue
    nodeVec.at(s) = std::make_tuple(0, null, true);
    auto cmp = [=] (tuple<int, double> a, tuple<int, double> b) -> bool {
        return std::get<1>(a) > std::get<1>(b);
    }; // the lambda function to compare tuple<nodeIndex, key> by key
    priority_queue<tuple<int, double>, vector<tuple<int, double>>,
decltype(cmp)> nodeQ(cmp); // tuple<nodeIndex, key>
    nodeQ.push(std::make_tuple(s, 0));
    while (!nodeQ.empty()) {
        tuple<int, double> node = nodeQ.top();
        nodeQ.pop();
        int u = std::get<0>(node);
        // if u should not be in Q, just pass
        if (std::get<2>(nodeVec.at(u)) == false) {
            continue;
        }
        std::get<2>(nodeVec.at(u)) = false; // set isInQ false because the
node has been included in the tree

        for (int j = 0; j < graph.at(u).size(); j++) {
            int v = graph.at(u).at(j).first;
            double length = graph.at(u).at(j).second;

            if (std::get<0>(nodeVec.at(u)) + length <
std::get<0>(nodeVec.at(v))) {
                std::get<0>(nodeVec.at(v)) = std::get<0>(nodeVec.at(u)) +
length;
                std::get<1>(nodeVec.at(v)) = u;
                nodeQ.push(std::make_tuple(v, std::get<0>(nodeVec.at(v))));
            }
        }
    }

    vector<pair<int, double>> result = vector<pair<int, double>>();
    for (int i = 0; i < nodeVec.size(); i++) {
        result.push_back(std::make_pair(std::get<1>(nodeVec.at(i)),
std::get<0>(nodeVec.at(i))));
    }
    return result;
}
```

附录 B 问题一最优路线求解-Python 源代码

```
# create a model
MODEL = Model()
# MODEL.setParam('OutputFlag', 0)

# add variables
x = MODEL.addVars(dist.keys(), obj=dist, vtype=GRB.BINARY, name='x')
d = MODEL.addVar(name="d")
MODEL.update()

# set the objective
MODEL.setObjective(d, GRB.MINIMIZE)

# add constraints
MODEL.addConstrs(quicksum(x[i, j, k] for j in range(V_NUM) for k in
range(K_NUM)) == 1 for i in node_list)
MODEL.addConstrs(quicksum(x[i, j, k] for j in range(V_NUM)) - quicksum(x[j,
i, k] for j in range(V_NUM)) == 0 for i in range(V_NUM) for k in
range(K_NUM))
MODEL.addConstrs(quicksum(x[ORIGIN_IDX - 1, j, k] for j in range(V_NUM)) ==
1 for k in range(K_NUM))
MODEL.addConstrs(d - quicksum(distance[i][j] / 1.5 * x[i, j, k] for i in
range(V_NUM) for j in range(V_NUM)) >= 0 for k in range(K_NUM))
MODEL.addConstrs(x[i, i, k] == 0 for i in range(V_NUM) for k in
range(K_NUM))
MODEL.addConstrs(x[i, j, k] + x[j, i, k] <= 1 for i in range(V_NUM) for j in
range(V_NUM) for k in range(K_NUM))

# callback - use lazy constraints to eliminate sub-tours
def mycallback(model, where):
    if where == GRB.Callback.MIPSOL:
        vals = model.cbGetSolution(model._vars[:-1])
        edges = list((i, j, k) for i, j, k in dict_linear.keys() if
vals[dict_linear[(i, j, k)]] > 0.5)
        shortest_cycle, k = cycle(edges)
        if shortest_cycle is not None:
            model.cbLazy(quicksum(x[i, j, k] for i in shortest_cycle for j in
shortest_cycle) <= len(shortest_cycle) - 1)


# find the loops
def cycle(edges):
    node_vehicle = {}
```

```
    G = nx.DiGraph()
    for e in edges:
        G.add_edge(e[0], e[1])
        node_vehicle[e[0]] = e[2]
        node_vehicle[e[1]] = e[2]
    shortest_subtour = None
    min = V_NUM
    cycle: list
    for cycle in nx.simple_cycles(G):
        if cycle.count(ORIGIN_IDX - 1) == 0:
            if len(cycle) < min:
                min = len(cycle)
                shortest_subtour = cycle
    if shortest_subtour is None:
        k = None
    else:
        k = node_vehicle[shortest_subtour[0]]
    return shortest_subtour, k


MODEL._vars = MODEL.getVars()
MODEL.Params.lazyConstraints = 1
MODEL.optimize(mycallback)
```

附录 C 问题二模型一求解-部分[2]Python 源代码

```
node_time =
[9.03,1.8,7.32,7.36,4.2,7.2,15.35,7.52,7.56,15.42815,5.73,7.68,4.2,25.4,25.5
,12,12.2008,25.8,15,24.948,5.67]

# add constraints
MODEL.addConstrs(quicksum(x[i, j, k] for j in range(V_NUM) for k in
range(K_NUM)) == 1 for i in node_list)
MODEL.addConstrs(quicksum(x[i, j, k] for j in range(V_NUM)) - quicksum(x[j,
i, k] for j in range(V_NUM)) == 0 for i in range(V_NUM) for k in
range(K_NUM))
MODEL.addConstrs(quicksum(x[ORIGIN_IDX - 1, j, k] for j in range(V_NUM)) ==
1 for k in range(K_NUM))
MODEL.addConstrs(d - quicksum(distance[i][j] / 1.5 * x[i, j, k] +
node_time[j] * x[i, j, k] for i in range(V_NUM) for j in range(V_NUM)) >= 0
for k in range(K_NUM))
```

[2] 其余部分大多与附录 B 重复。

```
MODEL.addConstrs(x[i, i, k] == 0 for i in range(V_NUM) for k in
range(K_NUM))
MODEL.addConstrs(x[i, j, k] + x[j, i, k] <= 1 for i in range(V_NUM) for j in
range(V_NUM) for k in range(K_NUM))
```

附录 D 问题二模型二求解-部分 Python 源代码

```
# failure occurrence
# V_left = [1, 2, 5, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
V_left = [1, 2, 5, 9, 10, 15, 16]
M_IDX = V_left.index(16) + 1
N_IDX = V_left.index(15) + 1
ORIGIN_IDX = V_left.index(ORIGIN_IDX) + 1
SERVICE_TIME = 600

V_left = [e-1 for e in V_left]
V_NUM = len(V_left)
V_l_0 = list(range(V_NUM))
V_l_0.remove(ORIGIN_IDX - 1)
V_l_0_m_n = list(range(V_NUM))
V_l_0_m_n.remove(M_IDX - 1)
V_l_0_m_n.remove(N_IDX - 1)
V_l_0_m_n.remove(ORIGIN_IDX - 1)
V_l_m_n = list(range(V_NUM))
V_l_m_n.remove(M_IDX - 1)
V_l_m_n.remove(N_IDX - 1)

dist = {(i, j, k): distance[V_left[i]][V_left[j]] for i in range(V_NUM) for
j in range(V_NUM) for k in range(K_NUM)}
dict_linear = {(i, j, k): i*V_NUM*K_NUM + j*K_NUM + k for i in range(V_NUM)
for j in range(V_NUM) for k in range(K_NUM)}

# create a model
MODEL = Model()
# MODEL.setParam('OutputFlag', 0)

# add variables
x = MODEL.addVars(dist.keys(), obj=dist, vtype=GRB.BINARY, name='x')
d = MODEL.addVar(name="d")
MODEL.update()

# set the objective
MODEL.setObjective(d, GRB.MINIMIZE)
```

```
# add constraints
MODEL.addConstrs(quicksum(x[i, j, k] for j in range(V_NUM) for k in
range(K_NUM)) == 1 for i in V_l_0)
MODEL.addConstrs(quicksum(x[i, j, k] for j in range(V_NUM)) - quicksum(x[j,
i, k] for j in range(V_NUM)) == 0 for i in V_l_0_m_n for k in range(K_NUM))
MODEL.addConstr(quicksum(x[i, j, k] for i in range(V_NUM) for j in [M_IDX -
1, N_IDX - 1] for k in range(K_NUM)) == 0)
MODEL.addConstrs(d - SERVICE_TIME * quicksum(x[M_IDX - 1, j, k] for j in
range(V_NUM))
              - quicksum(distance[V_left[i]][V_left[j]] / 1.5 * x[i, j, k]
                       + node_time[V_left[j]] * x[i, j, k] for i in
range(V_NUM) for j in range(V_NUM)) >= 0 for k in range(K_NUM))
MODEL.addConstrs(x[i, i, k] == 0 for i in range(V_NUM) for k in
range(K_NUM))
MODEL.addConstrs(x[i, j, k] + x[j, i, k] <= 1 for i in range(V_NUM) for j in
range(V_NUM) for k in range(K_NUM))
```

附录 E 问题三最优选址点求解-部分 Python 源代码

```
dmin = 201.24197185085075
origin_list = [2,6,1,5,4,3,7,11,10,19,9,16,20,8,21,14,17,15,18,13,12]
for o in origin_list:
    ORIGIN_IDX = o
    node_list = list(range(ORIGIN_IDX - 1)) + list(range(ORIGIN_IDX, V_NUM))

    # create a model
    MODEL = Model()
    # MODEL.setParam('OutputFlag', 0)

    # add variables
    x = MODEL.addVars(dist.keys(), obj=dist, vtype=GRB.BINARY, name='x')
    d = MODEL.addVar(name="d")
    MODEL.update()

    # set the objective
    MODEL.setObjective(d, GRB.MINIMIZE)

    # add constraints
    MODEL.addConstrs(quicksum(x[i, j, k] for j in range(V_NUM) for k in
range(K_NUM)) == 1 for i in node_list)
    MODEL.addConstrs(
```

```
        quicksum(x[i, j, k] for j in range(V_NUM)) - quicksum(x[j, i, k] for
j in range(V_NUM)) == 0 for i in
        range(V_NUM) for k in range(K_NUM))
    MODEL.addConstrs(quicksum(x[ORIGIN_IDX - 1, j, k] for j in range(V_NUM))
== 1 for k in range(K_NUM))
    MODEL.addConstrs(
        d - quicksum(distance[i][j] / 1.5 * x[i, j, k] for i in range(V_NUM)
for j in range(V_NUM)) >= 0 for k in
        range(K_NUM))
    MODEL.addConstrs(x[i, i, k] == 0 for i in range(V_NUM) for k in
range(K_NUM))
    MODEL.addConstrs(x[i, j, k] + x[j, i, k] <= 1 for i in range(V_NUM) for j
in range(V_NUM) for k in range(K_NUM))
    MODEL.addConstr(d <= dmin)

    MODEL._vars = MODEL.getVars()
    MODEL.Params.lazyConstraints = 1
    MODEL.optimize(mycallback)

    try:
        d_val = MODEL.getVarByName('d').x
        if d_val < dmin:
            dmin = d_val
        print(ORIGIN_IDX, MODEL.getVarByName('d'))
    except AttributeError:
        pass
```

附录 F 问题四最优路线求解-Python 源代码

```
# define the constants
V_NUM = 8
K_NUM = 3
L_NUM = 3
ORIGIN_IDX = 4
MAX = 1e5

# load and generate basic data
f = open("../data/distance", mode="rb")
distance = pickle.load(f)
f.close()
v = range(V_NUM)
ORIGIN_IDX = v.index(ORIGIN_IDX) + 1
node_list = list(range(ORIGIN_IDX - 1)) + list(range(ORIGIN_IDX, V_NUM))
```

```
dist = {(i, j, k, l): distance[v[i]][v[j]] for i in range(V_NUM) for j in
range(V_NUM) for k in range(K_NUM) for l in range(L_NUM)}
dict_linear = {(i, j, k, l): i*V_NUM*K_NUM*L_NUM + j*K_NUM*L_NUM + k*L_NUM +
l
            for i in range(V_NUM) for j in range(V_NUM) for k in
range(K_NUM) for l in range(L_NUM)}

# create a model
MODEL = Model()
# MODEL.setParam('OutputFlag', 0)

# add variables
x = MODEL.addVars(dist.keys(), obj=dist, vtype=GRB.BINARY, name='x')
d = MODEL.addVars(L_NUM, name="d")
MODEL.update()

# set the objective
MODEL.setObjective(quicksum(d[l] for l in range(L_NUM)), GRB.MINIMIZE)

# add constraints
MODEL.addConstrs(quicksum(x[i, j, k, l] for j in range(V_NUM) for k in
range(K_NUM) for l in range(L_NUM)) == L_NUM for i in node_list)
MODEL.addConstrs(quicksum(x[i, j, k, l] for j in range(V_NUM) for l in
range(L_NUM)) <= 1 for i in node_list for k in range(K_NUM))
MODEL.addConstrs(quicksum(x[i, j, k, l] for j in range(V_NUM)) -
quicksum(x[j, i, k, l] for j in range(V_NUM)) == 0
             for i in range(V_NUM) for k in range(K_NUM) for l in
range(L_NUM))
MODEL.addConstrs(quicksum(x[ORIGIN_IDX - 1, j, k, l] for j in range(V_NUM)
for k in range(K_NUM)) == 2 for l in range(L_NUM))
MODEL.addConstrs(d[l] - quicksum(distance[v[i]][v[j]] / 1.5 * x[i, j, k, l]
for i in range(V_NUM) for j in range(V_NUM)) >= 0
             for k in range(K_NUM) for l in range(L_NUM))
MODEL.addConstrs(x[i, i, k, l] == 0 for i in range(V_NUM) for k in
range(K_NUM) for l in range(L_NUM))
MODEL.addConstrs(x[i, j, k, l] + x[j, i, k, l] <= 1 for i in range(V_NUM)
for j in range(V_NUM) for k in range(K_NUM) for l in range(L_NUM))

# callback - use lazy constraints to eliminate sub-tours
def mycallback(model, where):
    if where == GRB.Callback.MIPSOL:
        vals = model.cbGetSolution(model._vars)
        cycles = []
        for l in range(L_NUM):
```

```
            for k in range(K_NUM):
                edges = list((i, j, k, l) for i, j, _, _ in dict_linear.keys()
if vals[dict_linear[(i, j, k, l)]] > 0.5)
                if cycles.count(edges) == 0:
                    cycles += cycle(edges)
        if len(cycles) != 0:
            for l in range(L_NUM):
                for k in range(K_NUM):
                    for c in cycles:
                        model.cbLazy(quicksum(x[i, j, k, l] for i in c for j in
c) <= len(c) - 1)


# find the loops
def cycle(edges):
    G = nx.DiGraph()
    for e in edges:
        G.add_edge(e[0], e[1])
    cycles = []
    cycle: list
    for cycle in nx.simple_cycles(G):
        if cycle.count(ORIGIN_IDX - 1) == 0:
            cycles.append(cycle)
    return cycles


MODEL._vars = MODEL.getVars()
MODEL.Params.lazyConstraints = 1
MODEL.optimize(mycallback)
```