中国研究生创新实践系列大赛

“华为杯”第二十一届中国研究生数学建模竞赛

学　　校		杭州电子科技大学
参赛队号		**23103360085**
队员姓名	**1.**	张丁鹏
	2.	欧阳利鑫
	3.	杨鸿铭

中国研究生创新实践系列大赛
“华为杯”第二十一届中国研究生数学建模竞赛

题 目： **自动化仓储系统智能搬运问题**

摘　　要：

自动化仓储系统在现代供应链管理中扮演着至关重要的角色，随着业务模式的多样化与市场的发展，传统的仓储系统已难以满足快速物流的需求。企业面临存储密度低、人工效率不足等问题，需要更高效的物流解决方案。因此，寻求一种高响应速度，快速货物寻找运输的自动化仓储系统具有重要意义。

针对问题一，本文首先进行了数据预处理，获取附件数据并根据曼哈顿距离和任务数量估算出了一个可能的时间步上限。在此基础上，建立了最小化运输时间的**整数线性规划模型**，并利用 **Gurobi** 求解器进行计算。在小规模地图（8x8 和 16x16）中，Gurobi 求解器成功求解并获得了最优解，但在大规模地图（64x64）上，求解器在合理时间内未能得出结果。为应对大规模问题，本文进一步采用了**改进的 GCBS 算法**，将最大完工时间和发生冲突的机器人对数量之和作为全局代价加速冲突解决过程。经过计算，改进的 GCBS 算法在不同规模地图上的求解结果如下：在 8x8 地图上，最优值为 **9**，求解时间为 0.0030 秒；在 16x16 地图上，最优值为 **21**，求解时间为 **0.04520** 秒；在 64x64 地图上，最优值为 **119**，求解时间为 **5.6734** 秒。其中改进的 GCBS 算法复杂度为$O(2^C \times b^d)$。

针对问题二，我们调整模型约束条件，以适应机器人到达终点后仍然占据终点位置的情况。首先，利用 Gurobi 求解器对调整后的**整数线性规划模型**进行计算。结果显示，在 8x8 和 16x16 的地图上，**Gurobi** 求解器成功获得了最优解，求解时间分别为 **2.7610 秒**和 **16.7350 秒**；但在 64x64 的地图上，求解器未能在合理时间内得出结果。随后，本文采用了**改进的 GCBS 算法**，并得出了以下结果：在 8x8 地图上，最优值为 **9**，求解时间为 **0.0010 秒**；在 16x16 地图上，最优值为 **21**，求解时间为 **0.0114 秒**；在 64x64 地图上，最优值为 **119**，求解时间为 **1.7826 秒**。该问题使用的算法与问题一算法一致故算法复杂度相同。

针对问题三，本文在问题二的基础上进一步扩展，考虑了机器人数量与任务总数不对等的情况。在这一设定下，建立了最小化运输时间的**整数线性规划模型**，并通过 A* 算法计算求出机器人起点以及各个任务终点到各个任务起点的距离矩阵，然后通过动态

规划为机器人分配任务，并在路径规划阶段应用改进的 GCBS 算法以求解具体路径。最终得出以下结果：在 8x8 地图上，最优值为 **65**，求解时间为 **0.0020 秒**；在 16x16 地图上，最优值为 **201**，求解时间为 **0.0689 秒**；在 64x64 地图上，最优值为 **705**，求解时间为 **1.8822 秒**。问题三使用了基于动态规划与改进 GCBS 算法的求解，其中算法复杂度为$O(m^2 \times 2^m + 2^c \times b^d)$。

针对问题四，本文在问题三的基础上，进一步引入了任务分配约束条件，即部分任务已被指定给特定的机器人来完成。为了解决这一问题，本文对动态规划与改进的 GCBS 算法进行了相应调整。具体来说，每个机器人只能选择其允许集合中的任务，并在任务分配和路径规划过程中始终满足这些约束条件。改进后的算法在不同规模地图上的求解结果如下：在 8x8 地图上，最优值为 **38**，求解时间为 **24.3966 秒**；在 16x16 地图上，最优值为 **124**，求解时间为 **0.0033 秒**；在 64x64 地图上，最优值为 **781**，求解时间为 **0.8403 秒**。问题四算法复杂度与问题三相同。

关键词：整数线性规划，Gurobi，GCBS 算法，动态规划，MAPF，NP-hard

目录

一、问题重述

1.1 问题的背景

自动化仓储系统在现代供应链管理中扮演着重要角色，尤其是在第三方物流（3PL）、鞋服行业、电商和 3C 制造等领域。随着业务模式的多样化和市场的发展，传统仓储系统难以满足日益增长的快速物流需求。当前，3PL 企业面临着存储密度低、人工效率不足的问题，因此，开发能够快速响应、精准执行货物运输任务的自动化仓储系统变得非常重要。这种系统的关键在于如何通过智能算法优化运输路径和时间，以最小化运输时间，提高整体运营效率。

1.2 问题的提出

设想一个工厂场景，其中工厂区域被划分为一个二维网格。在这个网格中，每个单元格都具有两种属性：可通行或不可通行（即障碍物）。在可通行的单元格中，有部分位置由机器人占据（在图中用标号的圆圈表示）。每个机器人都有两个位置属性，即其起点和终点位置（在图中用方块表示）。我们的目标是在这个场景中，通过合理规划，以最小化运输总时间来完成所有给定的运输任务

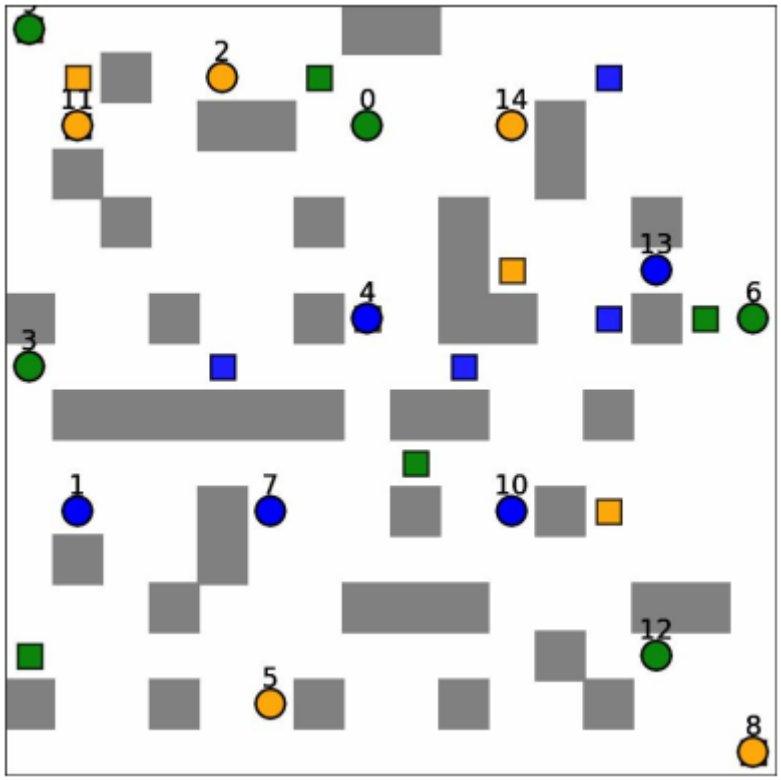

图 1-1 智能搬运路径示意图

问题一：在给定的网格场景中，假设每个机器人只有一个任务目标，并且在完成任务后会从系统中消失。要求在不发生碰撞的前提下，建立一个数学模型，以最小化所有机器人完成任务的总运输时间。并对提供的数据集计算所有机器人的总运输时间、路径方案及算法复杂度进行分析。

问题二：基于问题1的基础，假设每个机器人在完成任务后停留在目的地而不消失。在此条件下，要求建立一个最小化总运输时间的数学模型，并对提供的数据集计算总运输时间、路径方案及算法复杂度进行分析。

问题三：在问题2的基础上，假设机器人数量与任务总数不对等（任务多于或少于机器人数量）。要求在不发生冲突的前提下，构建一个任务调度的数学模型，以确保任务总运输时间最小化。并对提供的数据集计算总运输时间、路径方案及算法复杂度进行分析。

问题四：在问题3的基础上，假设部分任务已经被指定给特定的机器人，并且任务的完成顺序不固定。在不发生冲突的前提下，要求构建一个数学模型，并设计调度算法，以确保任务总运输时间最小化。然后对提供的数据集计算总运输时间、路径方案及算法复杂度进行分析。

二、符号说明和基本假设

3.1 基本假设

1、不考虑机器人取货和卸货的时间

2、相邻的网格仅为前后左右四个网格，对角线上的相邻网格不计入相邻网格

3、每个时间步机器人只能移动 1 格或者不动

3.2 符号说明

符号	说明
m	机器人数量
R	机器人集合$R = \{i = 1,2 \dots m\}$
n	地图上能到达的点的数量
S	地图中可到达的坐标集合为$S = \{j = 1,2,3 \dots n\}$
q_i	各机器人的起点坐标q_i
z_i	各任务终点坐标z_i
δ_j	表示经过一个事件步长能由j点到达的点的集合
u	机器人的时间步上限
T	机器人的时间步集合$T = \{t = 1,2 \dots u\}$;约定机器人在起点时时间步为 1
K	机器人任务集合$K = \{k = 1,2 \dots p\}$
p	任务总数
ϑ_i	表示机器人i已经分配的任务集合
o_i	机器人i的初始位置
x_{ijt}	机器人i在时间t是否在位置j
τ	最大完成时间
y_{ik}	机器人i是否做任务k
b_{ikt}	机器人i在t时间是否取k任务的货
c_{ikt}	机器人i在t时间是否放下k任务的货物

注：其他变量会在文章用到时说明

三、问题一

3.1 问题一的分析

问题一的目标是在机器人完成任务后消失的前提下最小化所有机器人完成任务的最大完工时间，同时确保机器人之间不发生碰撞。为避免机器人之间的冲突，需要设置约束条件。通过查阅文献[1][3]，此类问题冲突类型总结如图 3-1 本题应确保在任何时间步内，两个机器人不会同时占据同一位置（顶点冲突）或通过同一条边（交换冲突），对于边冲突、追随冲突可以通过机器人一个时间最多只能在一个坐标点上这一约束控制，循环冲突通过最小化完成时间的目标控制。基于以上分析，本题可建立整数线性规划模型，通过求解器和设计相关算法进行求解以及设计改进的 GCBS 算法进行求解。

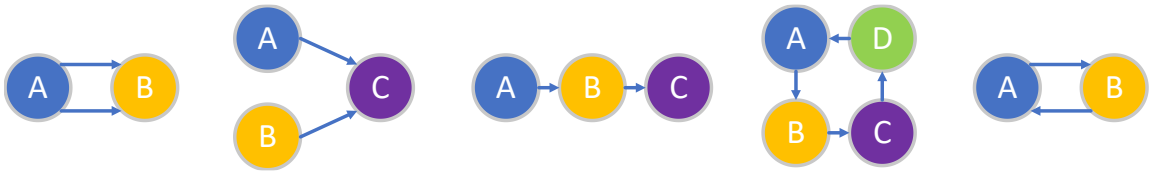

图 3-1 常见冲突类型示意图。从左到右依次为:边冲突、顶点冲突、追随冲突、循环冲突以及交换冲突

3.2 数据预处理

通过对附件数据的观察分析，本题需要从附件中获取地图信息、机器人数量、机器人起点、任务数量和各个任务的起点和终点数据。我们利用 python 获得位置、邻域等数据，并计算出了一个可能的时间步上限，各附件数据见表 3-1。

表 3-1 附件三部分数据

名称	符号	8x8map	16x16map	64x64map
机器人数量	m	8	12	60
位置数量	n	58	161	3277
估算时间步上限	u	17	25	179
任务数量	p	8	12	60

其中时间步上限为不考虑障碍物情况下（此时两点之间时间步可通过曼哈顿距离计算）每个机器人完成任务所需时间的最大值+任务数量，只有机器人从起点到终点连成的直线上出现大量连续障碍时才会使上限失效，这是一个很难达到的条件，因此以此为估算的时间步上限。

3.3 整数线性规划模型的建立

3.3.1 定义决策变量

1、机器人i在时间t是否在位置j

$$x_{ijt}=\begin{cases}1;\text{机器人}i\text{在 j 时间步上在坐标点 j 上}\\0;\text{否则}\end{cases};i\in R,j\in S,t\in T \qquad (3-1)$$

2、机器人的运输总时间

$$\tau \geq 0 \tag{3-2}$$

3.3.2 目标与约束条件设置

1、约束条件

（1）开始时每个机器人都在各自起点

$$x_{iq_i1} = 1; i \in R \tag{3-3}$$

其中q_i表示机器人i的起点。

（2）机器人必须且仅到达目标点一次

由题目已知条件机器人到达终点后消失，因此每个机器人必须且仅能到达目标点一次。

$$\sum_{\forall t \in T} x_{iz_it} = 1; i \in R \tag{3-4}$$

其中z_i表示机器人i的终点。

（3）机器人一个时间最多只能在一个坐标点上

$$\sum_{\forall j \in S} x_{ijt} \leq 1; i \in R, t \in T \tag{3-5}$$

（4）两个机器人不能同时在一处（边冲突、顶点冲突）

$$x_{i_1jt} + x_{i_2jt} \leq 1; i_1, i_2 \in R, i_1 \neq i_2, j \in S, t \in T \tag{3-6}$$

（5）交换冲突

交换冲突表示两个机器人在邻近的时间步内不能交换位置，当且仅当$x_{i_1j_1t-1}, x_{i_1j_2t}, x_{i_2j_2t-1}, x_{i_2j_1t}$值全为 1 时才会出现此冲突，因此设置约束为以上 4 个决策变量的和小于等于 3；并且只有在j_1, j_2互为邻域时才会出现交换冲突。基于上述分析建立以下交换冲突约束：

$$x_{i_1j_1t-1} + x_{i_1j_2t} + x_{i_2j_2t-1} + x_{i_2j_1t} \leq 3; i_1, i_2 \in R, i_1 \neq i_2, j_1 \in S, j_2 \in \delta_{j_1}, t \in T \backslash \{1\} \tag{3-7}$$

其中δ_{j_1}表示j_1的邻域，即j_1点一个时间步后所能达到的点集合，并且$j \notin \delta_j$。

（6）机器人运动范围约束

每个机器人在一个时间步长后只能到达其前后左右的四个点，因此每个机器人下一个位置只能在其当前位置的邻域内，或者停在原地不动。但是对于本题又有机器人到达终点后会消失，所以当机器人到达终点后下一时间步长后不用满足这个约束。综上建立本题的机器人运动范围约束：

$$x_{ijt-1} \leq \sum_{\forall j' \in \delta_j \cup \{j\}} x_{ij't}; i \in R, j \in S, j \neq z[i], t \in T \backslash \{1\} \tag{3-8}$$

（7）到达终点后消失

当机器人到达终点后，其后时间步内该机器人不会出现在任何一个坐标点上。

$$\sum_{t'=t+1}^{u} x_{ijt'} \leq \left(1 - x_{iz_it}\right) \cdot Inf; i\epsilon R, t\epsilon T\backslash\{u\} \quad (3-9)$$

其中Inf表示一个无穷大的正数。

（8）运输总时间计算

最大到达时间由各机器人到达终点所用时间确定

$$\tau \geq (t-1) \cdot x_{iz_it}; i \in R, t \in T \quad (3-10)$$

2、目标函数

目标函数为最小化运输总时间，由于以此为目标循环冲突约束不用额外加上约束。

$$Min\ \tau \quad (3-11)$$

3.3.3 整体整数线性规划模型

$$\begin{cases} Min\ \tau \\ x_{iq_i1} = 1; i \in R \\ \sum_{\forall t \in T} x_{iz_it} = 1; i \in R \\ \sum_{\forall j \in S} x_{ijt} \leq 1; i \in R, t \in T \\ x_{i_1jt} + x_{i_2jt} \leq 1; i_1, i_2 \in R, i_1 \neq i_2, j \in S, t \in T \\ x_{ijt-1} \leq \sum_{\forall j' \in \delta_j \cup \{j\}} x_{ij't}; i \in R, j \in S, j \neq z[i], t \in T\backslash\{1\} \\ \sum_{t'=t+1}^{u} x_{ijt'} \leq \left(1 - x_{iz_it}\right) \cdot Inf; i\epsilon R, t\epsilon T\backslash\{u\} \\ \tau \geq (t-1) \cdot x_{iz_it}; i \in R, t \in T \\ x_{ijt} \in \{0,1\}; i \in R, j \in S, t \in T \\ \tau \geq 0 \end{cases}$$

3.4 问题求解

3.4.1 使用 Gurobi 求解器求解

上文建立了问题 1 的整数线性规划模型，整数规划模型可以利用商业求解器进行求解，因此我们设计了基于 Gurobi 的求解方案对附件一中三个地图进行了求解，结果显示前两个地图求解器均很快求出了最优解，求出任务总运输时间最小分别为 **9** 和 **21**，但由于第三个问题规模太大，求解器无法在很短时间内求出结果。求解器的求解结果见表 3-2

表 3-2 基于 Gurobi 求解器的求解结果

地图	最优值	Gap	求解时间
8*8	9.0	0	2.0339
16*16	21.0	0	14.1500
64*64	-	-	-

注：$Gap = \frac{obj-lb}{obj}$,其中obj为模型在 600s 内求解的目标函数值，lb为最优值的下界。$Gap = 0$则表示当前求解的目标函数值为最优值。

3.4.2 基于改进的 GCBS 算法求解

1、改进的 GCBS 算法设计

为了解决整数线性规划模型在求解复杂地图问题时运行时间过长的问题，我们采用了 GCBS（Greedy Conflict-Based Search，贪心冲突基搜索）。GCBS 是在 CBS 的基础上通过引入贪心策略加速冲突解决过程。

具体而言，在改进的 GCBS 算法中，全局代价被定义为最大完工时间和发生冲突的机器人对（pairs）数量之和。

使用 GCBS 的算法求解最小机器人总运输时间的流程图如下：

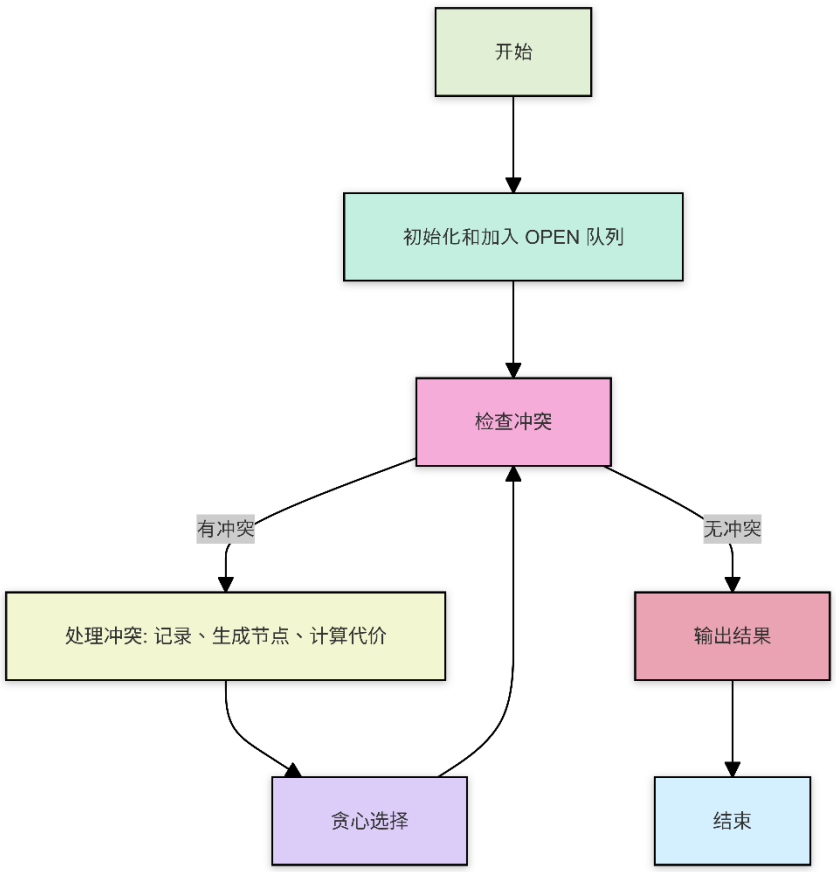

图 3-2GCBS 算法流程图

以下是算法执行步骤

1.初始化：设定初始状态，将所有机器人的路径设置为从起点到终点的最短路径（使用 A*算法计算）。将初始节点添加到优先队列`OPEN`中，优先级按照代价函数（全局代价）进行排序。

2.冲突检测：对`OPEN`队列中的节点，检查其对应的路径集合是否存在冲突（如两个机器人同时占据同一位置或沿相同边移动）。如果存在冲突，记录所有冲突的机器人对（pairs）。

3. 冲突解决：对每个检测到的冲突，分别生成两个新的子节点，每个节点对应一个不同的约束条件（例如约束其中一个机器人不能在冲突的时间点占据冲突位置）。重新计算这两个节点的代价并将其添加到`OPEN`队列中。代价计算时考虑最大完工时间和冲突机器人对的数量。

4. 贪心选择：在冲突解决时，优先选择减少最大完工时间与冲突机器人对数量之和的约束方案（即贪心策略），以快速降低全局代价。

5.终止条件：当`OPEN`队列中存在一个无冲突的节点时，算法终止，该节点对应的路径集合即为最终的解决方案。如果`OPEN`队列为空且未找到无冲突的路径集合，则算法失败。

6.输出：输出最终的路径集合，以及对应的最大完工时间。

算法复杂度：$O(2^C \times b^d)$其中C是冲突的数量, b是分支因子,d是路径的深度。

2、改进的 GCBS 算法结果展示

三张地图的最小机器人运输总时间汇总如下表：

表 3-3 改进的 GCBS 算法结果

地图	最优值	求解时间
8*8	9.0	0.0030
16*16	21.0	0.04520
64*64	119	5.6734

从结果可以看出，改进的 GCBS 算法结果在与整数规划模型一致的情况下，求解时间小了两个数量级，这是因为整数线性规划考虑的情况数量远远多于 GCBS 模型。

3.5 结果展示

经过两种方法的求解，本题最终输出的机器人运输总时间和以及求解时间如表 3-4 所示：

表 3-4 各个地图机器人运输总时间和以及求解时间结果

地图	最优值	求解时间
8*8	**9.0**	0.0030
16*16	**21.0**	0.04520
64*64	**119**	5.6734

从表中可以看出，问题一地图 8*8 16*16 64*64 对应的机器人运输总时间和分别为**9,21,119**。完整求解结果见对应数据集附件 results_第一问.xls

为了更加直观地展示结果，本文绘制了三张地图的路径图，其中黑色圆表示障碍物，橘色圆和绿色圆分别表示起点和终点，不同颜色的线表示各个机器人的运行路径：

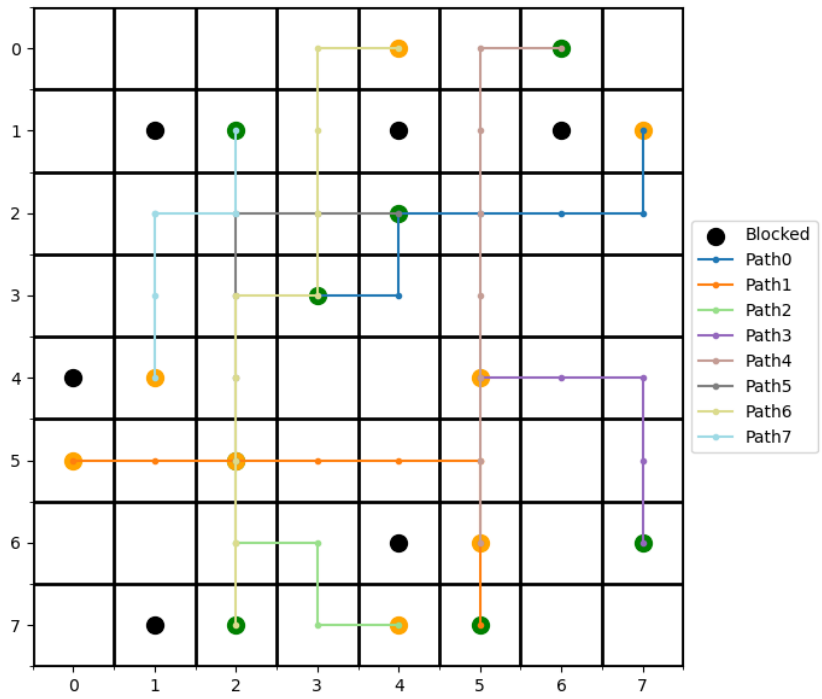

图 3-3 问题 1 _8*8 机器人路径图

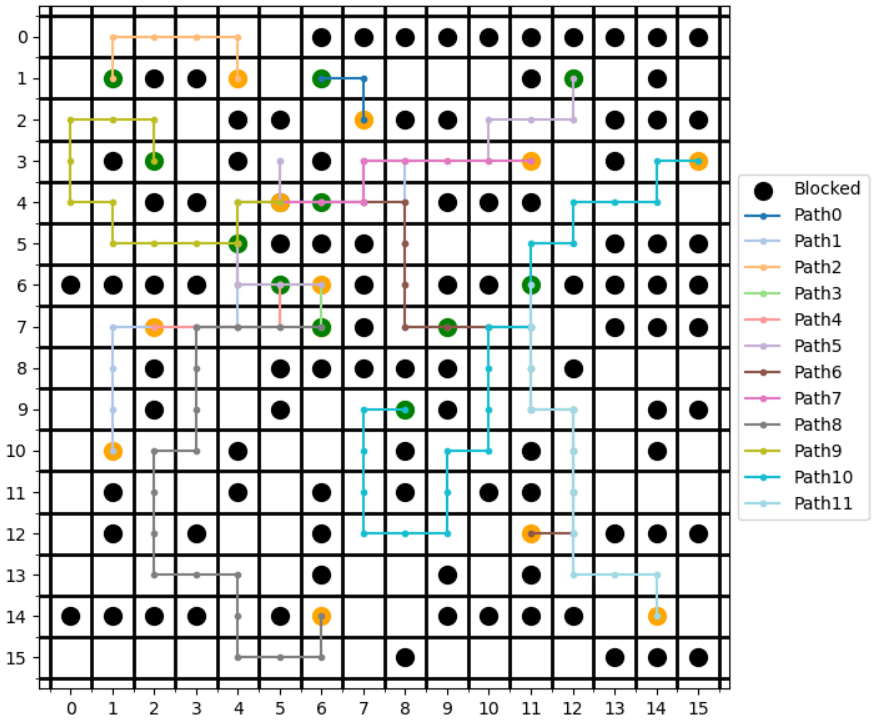

图 3-4 问题 1_16*16 机器人路径图

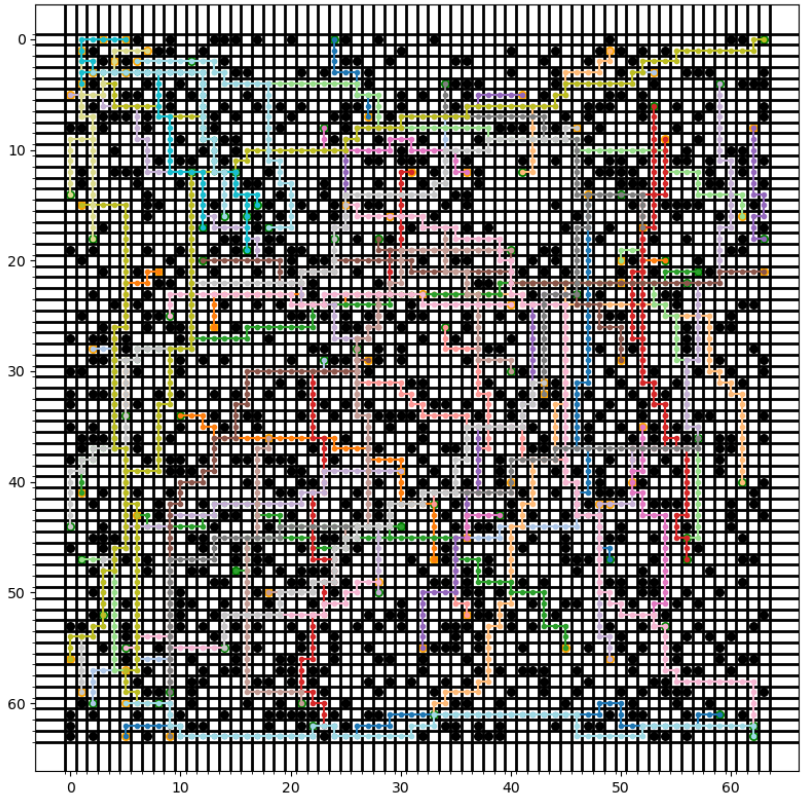

图 3-5 问题 1_64*64 机器人路径图

四、问题二

4.1 问题二的分析

问题二在第一问的基础上加入了机器人完成任务后停留在目的地而不消失的设定，所以模型在问题一的基础上调整了约束条件：机器人到达终点后继续占据终点位置，而其余条件保持不变。同样本题可以通过利用求解器与改进的 GCBS 算法求解。

4.2 基于问题一的整数线性规划模型的建立

4.2.1 模型建立

问题 2 的模型只需在问题 1 的基础上对部分约束条件进行修改即可，决策变量设置与目标函数保持不变，对于约束 2，6，7，8 进行调整，具体如下：

（1） 机器人必须到达终点

由于第 1 题机器人到达终点后消失，本题机器人到达终点后需要保持原地不动，即机器人在终点位置上的次数会超过一次。

$$\sum_{\forall t \in T} x_{iz_i t} \geq 1; i \in R \qquad (4-1)$$

这个约束较第一题是一个更弱的约束，会加大求解难度。

（2）机器人运动范围约束

本题机器人到达终点不消失，所以本约束不考虑第一题提到的例外情况。

$$x_{ijt-1} \leq \sum_{\forall j' \in \delta_j \cup \{j\}} x_{ij't}; i \in R, j \in S, t \in T \backslash \{1\} \quad (4-2)$$

（3）机器人到达终点保持不动

本题删除机器人到达终点后消失约束，新增到达终点后不再移动约束

$$x_{iz_it+1} \geq x_{iz_it}; i \epsilon R, t \epsilon T \backslash \{u\} \quad (4-3)$$

（4）运输总时间计算

$$\tau \geq u - \sum_{\forall t \in T} x_{iz_it}; i \in R \quad (4-4)$$

4.2.2 整体整数线性规划模型

$$Min\ \tau$$

$$\begin{cases} x_{iq_i1} = 1; i \in R \\ \sum_{\forall t \in T} x_{iz_it} \geq 1; i \in R \\ \sum_{\forall j \in S} x_{ijt} \leq 1; i \in R, t \in T \\ x_{i_1jt} + x_{i_2jt} \leq 1; i_1, i_2 \in R, i_1 \neq i_2, j \in S, t \in T \\ x_{ijt-1} \geq \sum_{\forall j' \in \delta_j \cup \{j\}} x_{ij't}; i \in R, j \in S, t \in T \backslash \{1\} \\ x_{iz_it+1} \geq x_{iz_it}; i \epsilon R, t \epsilon T \backslash \{u\} \\ \tau \geq u - \sum_{\forall t \in T} x_{iz_it}; i \in R \\ x_{ijt} \in \{0,1\}; i \in R, j \in S, t \in T \\ \tau \geq 0 \end{cases}$$

4.3 问题求解

4.3.1 基于 Gurobi 求解器求解

与第 1 题一样，我们首先设计了基于 Gurobi 求解器的解决方案，求解结果如表 4-1 所示，可以看出求解器依然只能求解前两个较小规模数据，且通过对比求解时间，问题 2 略长于问题 1，间接说明问题 2 的模型更难求解。

表 4-1 整数线性规划求解器结果

地图	最优值	Gap	求解时间
8*8	9.0	0	2.7610
16*16	21.0	0	16.7350
64*64	-	-	-

4.3.2 基于改进的 GCBS 算法求解

我们在问题 1 改进的 GCBS 算法的基础上，加入机器人到了终点后并不消失，而是停留在目标点不动的条件进行了求解，算法复杂度与问题一相同

表 4-2 改进的 GCBS 算法结果

地图	最优值	求解时间
8*8	9.0	0.0010
16*16	21.0	0.0114
64*64	119	1.7826

4.4 结果展示

经过两种方法的求解，本题最终输出的机器人运输总时间和以及求解时间如表 4-3 所示：

表 4-3 各个地图器人运输总时间和以及求解时间结果

地图	最优值	求解时间
8*8	9.0	0.0010
16*16	21.0	0.0114
64*64	119	1.7826

为了更加直观地展示结果，本文绘制了三张地图的路径图，其中黑色圆表示障碍物，橘色圆和绿色圆分别表示起点和终点，不同颜色的线表示各个机器人的运行路径：

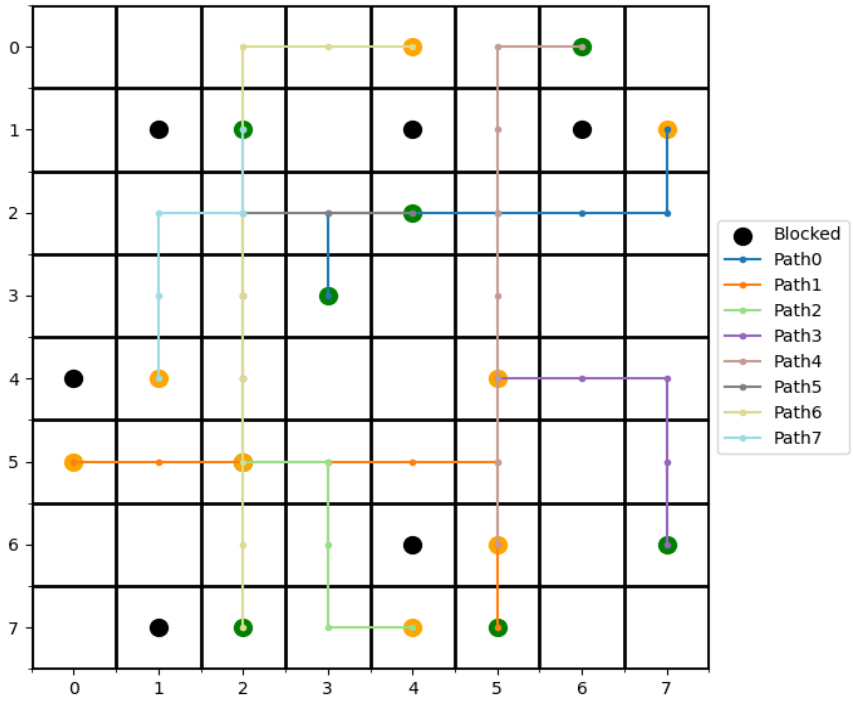

图 4-1 问题 2_8*8 机器人路径图

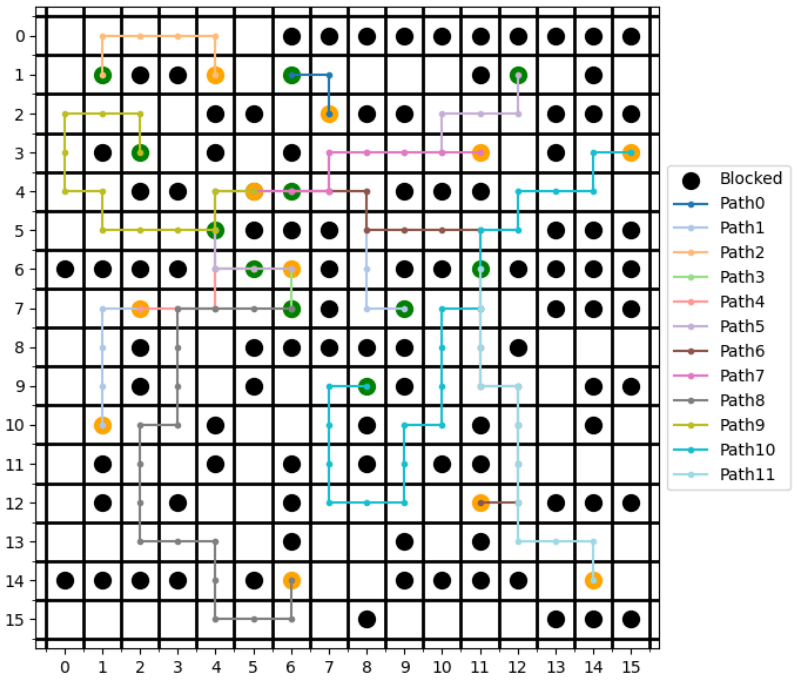

图 4-2 问题 2_16*16 机器人路径图

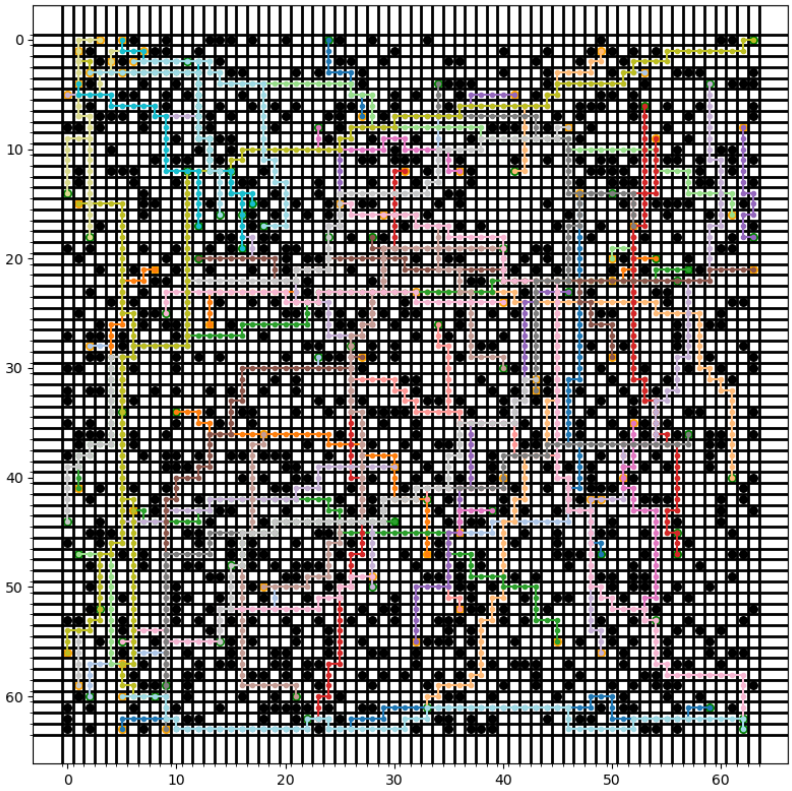

图 4-3 问题 2_64*64 机器人路径图

五、问题三

5.1 问题三的分析

问题 3 要求在问题 2 的基础上，假设机器人数目与任务总数不对等，在不发生冲突的前提下，构建数学模型完成机器人的任务调度且以确保任务总运输时间最小。因此首先需要将全部任务分配给机器人；其次，考虑到机器人需要到达任务起点取货送至任务终点，且机器人一次只能完成一个任务，所以需要决定机器人完成任务的顺序并确定机器人取货与送货的时刻以保证先取后送和送完才能再取的约束；最后还需要考虑机器人完成最后一个任务需要停留在原地，机器人可能不止一次经过各任务的起点和终点等情况，我们可以采用整数规划模型以及设计动态规划和改进的 GCBS 联合算法两种方法对附件进行求解

5.2 数据预处理

通过对附件数据的观察分析，本题需要从附件中获取地图信息、机器人数量、机器人起点、任务数量和各个任务的起点和终点数据。我们首先利用 python 对相关数据进行了获取，与问题 12 类似操作获得位置、邻域等数据，并计算出了一个可能的时间步上限，各附件数据见表 5-1。

表 5-1 附件三部分数据

名称	符号	8x8map	16x16map	64x64map
机器人数量	m	1	1	5
位置数量	n	58	161	3277
估算时间步上限	u	81	191	4771
任务数量	p	8	12	60

其中时间步上限为不考虑障碍物情况下（此时两点之间时间步可通过横纵坐标差值的绝对值之和计算）机器人 1 由起点按顺序完成所有任务所需步数，只有当机器人数量为 1 且最优方案就是按标号顺序完成所有任务且障碍物会印象运输的情况下这个上界才会失效，这是一个很难达到的条件，因此以此为估算的时间步上限。

5.3 整数线性模型的建立

5.3.1 定义决策变量

1、机器人 i 在时间 t 是否在位置 j

$$x_{ijt} = \begin{cases} 1; \text{机器人}i\text{在时间}t\text{处于位置}j \\ 0; \text{否则} \end{cases}; i \in R, j \in S, t \in T \qquad (5-1)$$

2、机器人 i 是否做任务 k

$$y_{ik=}\begin{cases}1;\text{机器人}i\text{做 k 任务}\\0;\text{否则}\end{cases};i\in R,k\in K \quad (5-2)$$

3、机器人 i 在 t 时间是否取k任务的货

$$b_{ikt}=\begin{cases}1;\text{机器人}i\text{在 t 时间取 k 任务的货}\\0;\text{否则}\end{cases};i\in R,k\in K,t\in T \quad (5-3)$$

4、机器人 i 在 t 时间是否放下k任务的货物

$$c_{ikt}=\begin{cases}1;\text{机器人}i\text{在 t 时间放下 k 任务的货}\\0;\text{否则}\end{cases};i\in R,k\in K,t\in T \quad (5-4)$$

5、最大完成时间

$$\tau\geq 0 \quad (5-6)$$

5.3.2 目标与约束条件设置

1、约束条件

第一部分：位置时间相关约束

（1）机器人从起点出发

$$x_{io_i1}=1;i\in R \quad (5-7)$$

其中o_i表示机器人i的起点

（2）机器人每个时间都处于唯一的位置

$$\sum_{\forall t\in T}x_{ijt}=1;i\in R,j\in S,t\in T \quad (5-8)$$

（3）两个机器人同一时刻不能处于同一位置

$$x_{i_1jt}+x_{i_2jt}\leq 1;i_1,i_2\in R,i_1\neq i_2,j\in S,t\in T \quad (5-9)$$

（4）交换冲突

$$x_{i_1j_1t-1}+x_{i_1j_2t}+x_{i_2j_2t-1}+x_{i_2j_1t}\leq 3;i_1,i_2\in R,i_1\neq i_2,j_1\in S,j_2\in\delta_{j_1},t\in T\backslash\{1\} \quad (5-10)$$

（5）机器人运动范围约束

$$x_{ijt-1}\leq\sum_{\forall j'\in\delta_j\cup\{j\}}x_{ij't};i\in R,j\in S,t\in T\{1\} \quad (5-11)$$

（6）每个任务都要分配给一个机器人

$$\sum_{\forall i\in R}y_{ik}=1;k\in K \quad (5-12)$$

第二部分：货物拿放约束

（7）只有选择了某任务才能在到达该任务起点取货，终点放下货物

$$b_{ikt}\leq\frac{1}{2}\left(x_{iq_kt}+y_{ik}\right);i\in R,t\in T,k\in K \quad (5-13)$$

$$c_{ikt} \le \frac{1}{2}\left(x_{iz_kt} + y_{ik}\right); i \in R, t \in T, k \in K \quad (5-14)$$

（8）同一个任务先拿货后送货

$$\sum_{\forall t \in T} t \cdot b_{ikt} \le \sum_{\forall t \in T} t \cdot c_{ikt}\ ;\ i \in R, k \in K \quad (5-15)$$

（9）所有任务必须进行一次取货放货

$$\sum_{\forall i \in R}\sum_{\forall t \in T} b_{ikt} = 1; k \in K \quad (5-16)$$

$$\sum_{\forall i \in R}\sum_{\forall t \in T} c_{ikt} = 1; k \in K \quad (5-17)$$

（10）机器人装载货物数量约束

机器人不能在没有拿到货物的情况下送货，也不能在上一个货物未送达的情况下取下一件货物。

$$0 \le \sum_{t'=1}^{t}\sum_{\forall k \in K} b_{ikt'} - c_{ikt'} \le 1; i \in R, t \in T \quad (5-18)$$

（11）机器人选择了几个任务就必须把完成相应次数的取货放货

$$\sum_{\forall k \in K}\sum_{\forall t \in T} b_{ikt} = \sum_{\forall k \in K} y_{ik}\ ; i \in R \quad (5-19)$$

$$\sum_{\forall k \in K}\sum_{\forall t \in T} c_{ikt} = \sum_{\forall k \in K} y_{ik}\ ; i \in R \quad (5-20)$$

（12）机器人完成所有任务后停在原地

$$x_{ijt} \le x_{ijt+1} + \left(\sum_{\forall k \in K} y_{ik} - \sum_{\forall k \in K}\sum_{t'=1}^{t} c_{ikt}\right); i \in R, j \in S, t \in T \backslash \{u\} \quad (5-21)$$

（12）求出机器人i完成所有任务的时间

$$\tau \ge (t-1) \cdot \sum_{\forall k \in K} c_{ikt}\ ; i \in R, t \in T \quad (5-22)$$

2、目标函数

目标函数为最小化所有任务完成时间

$$Min\ \tau \quad (5-23)$$

5.3.3 整体整数线性模型

$$Min\ \tau$$

$$s.t.\begin{cases} x_{io_i1} = 1; i \in R \\ \sum_{\forall t \in T} x_{ijt} = 1; i \in R, j \in S, t \in T \\ x_{i_1jt} + x_{i_2jt} \le 1; i_1, i_2 \in R, i_1 \ne i_2, j \in S, t \in T \\ x_{i_1j_1t-1} + x_{i_1j_2t} + x_{i_2j_2t-1} + x_{i_2j_1t} \le 3; i_1, i_2 \in R, i_1 \ne i_2, j_1 \in S, j_2 \in \delta_{j_1}, t \in T\backslash\{1\} \\ x_{ijt-1} \le \sum_{\forall j' \in \delta_j \cup \{j\}} x_{ij't}; i \in R, j \in S, t \in T\backslash\{1\} \\ \sum_{\forall i \in R} y_{ik} = 1; k \in K \\ b_{ikt} \le \frac{1}{2}\left(x_{iq_kt} + y_{ik}\right); i \in R, t \in T, k \in K \\ c_{ikt} \le \frac{1}{2}\left(x_{iz_kt} + y_{ik}\right); i \in R, t \in T, k \in K \\ \sum_{\forall t \in T} t \cdot b_{ikt} \le \sum_{\forall t \in T} t \cdot c_{ikt};\ i \in R, k \in K \\ \sum_{\forall i \in R} \sum_{\forall t \in T} b_{ikt} = 1; k \in K \\ \sum_{\forall i \in R} \sum_{\forall t \in T} c_{ikt} = 1; k \in K \\ 0 \le \sum_{t'=1}^{t} \sum_{\forall k \in K} b_{ikt'} - c_{ikt'} \le 1; i \in R, t \in T \\ \sum_{\forall k \in K} \sum_{\forall t \in T} b_{ikt} = \sum_{\forall k \in K} y_{ik}; i \in R \\ \sum_{\forall k \in K} \sum_{\forall t \in T} c_{ikt} = \sum_{\forall k \in K} y_{ik}; i \in R \\ x_{ijt} \le x_{ijt+1} + \left(\sum_{\forall k \in K} y_{ik} - \sum_{\forall k \in K} \sum_{t'=1}^{t} c_{ikt}\right); i \in R, j \in S, t \in T\{u\} \\ \tau \ge (t-1) \cdot \sum_{\forall k \in K} c_{ikt}; i \in R, t \in T \\ x_{ijt} \in \{0,1\}; i \in R, j \in S, t \in T \\ y_{ik} \in \{0,1\}; i \in R, k \in K \\ b_{ikt} \in \{0,1\}; i \in R, k \in K, t \in T \\ c_{ikt} \in \{0,1\}; i \in R, k \in K, t \in T \\ \tau \ge 0 \end{cases}$$

5.4 问题求解

5.4.1 基于 Gurobi 求解器求解

我们设计了基于 Gurobi 求解器求解本题模型的方案，输入附件三进行求解发现由于问题过于复杂，求解器无法在 1200s 内求解。但是通过降低问题规模，假设第一个数据中的任务只有前三个，求解器花费 43.178 秒求出了最优解，对解进行检查，没有发现异常，由此证明了我们建立模型的有效性。

5.4.2 基于动态规划与改进的 GCBS 算法的求解

在此部分，我们结合动态规划和改进的 GCBS 算法，求解多机器人路径规划问题。具体地，我们首先用 A*算法求出机器人到各个任务起点的距离，以及各个任务终点到各个任务起点的距离矩阵，然后通过动态规划求解机器人任务分配的问题，并在路径规划阶段应用改进的 GCBS 算法以优化整体路径。需要注意的细节是使用改进的 GCBS 算法解决冲突时，A*算法的起点和终点应该取决于机器人正在执行哪个任务，否则可能出现机器人没有执行某项任务的情况。

为了求解机器人在给定任务集合中的最优任务分配，我们采用动态规划（DP）方法。具体步骤如下：

1.状态定义：我们使用一个字典 `dp` 来记录不同状态下的最优解，`dp[state]` 表示在特定状态下每个机器人的任务总时间。`state` 是一个元组，每个元素表示对应机器人的任务集（通过位运算表示）。

2.初始化状态：初始状态为每个机器人可以选择任意一个任务作为第一个任务。我们通过遍历每个机器人 `k` 和任务 `j` 来初始化 `dp`，并计算初始任务的执行时间。

3.状态转移：在动态规划的主循环中，我们枚举所有可能的任务组合 `S`，并对每个机器人和任务执行状态转移。对于每个当前状态 `current_state` 和机器人 `k`，我们尝试将未分配的任务 `j` 添加到 `k` 的任务集中，并计算新的任务总时间。如果新的状态更优，则更新 `dp` 和路径 `path`。

4.最优解的计算：我们在所有可能的状态中搜索覆盖所有任务的最优分配方案，并返回最小的任务总时间 `final_result` 和对应的任务路径 `optimal_paths`。

在路径规划阶段，我们基于 GCBS 算法进行优化。该算法通过贪心策略优先解决对全局代价影响较大的冲突，并结合动态规划求得的任务分配方案，最终确定各机器人的最优路径。

1.任务分配：基于动态规划结果，确定每个机器人执行的任务顺序。

2.路径规划： 使用改进的 GCBS 算法，依照任务分配顺序规划机器人路径。算法在搜索过程中动态解决机器人之间的冲突，以最小化全局代价（最大完工时间和冲突对数量之和）。

算法复杂度： $O(m^2 \times 2^m + 2^C \times b^d)$,其中 m 是任务数。

5.4 结果展示

经过基于动态规划与改进的 GCBS 算法求解，各个地图器人运输总时间和以及求解时间结果如表 5-2 所示

表 5-2 各个地图机器人运输总时间和以及求解时间结果

地图	最优值	求解时间
8*8	**65.0**	0.0020
16*16	**201.0**	0.0689
64*64	**705**	1.8822

从表中可以看出，问题三地图 8*8 16*16 64*64 对应的任务总运输时间分别为 **65,201,705**。完整求解结果见对应数据集附件 results_第三问.xls。

其中每张地图中各个机器人执行任务的顺序如表 5-3 所示

表 5-3 各个地图器人执行任务顺序表

地图	机器人编号	任务顺序
8*8	**1**	[6, 1, 2, 5, 3, 4, 0, 7]
16*16	**1**	[2, 0, 7, 4, 5, 10, 8, 3, 1, 11, 6, 9]
64*64	**1**	[0, 41, 27, 3, 34, 2, 37, 21, 8, 54, 58, 57]
	2	[50, 6, 4, 9, 24, 17, 5, 47, 53, 29, 11]
	3	[39, 15, 14, 32, 48, 59, 20, 18, 43, 45, 55]
	4	[1, 23, 30, 44, 38, 12, 13, 28, 49, 26, 16, 56]
	5	[40, 7, 42, 25, 33, 46, 35, 10, 31, 19, 22, 36, 52, 51]

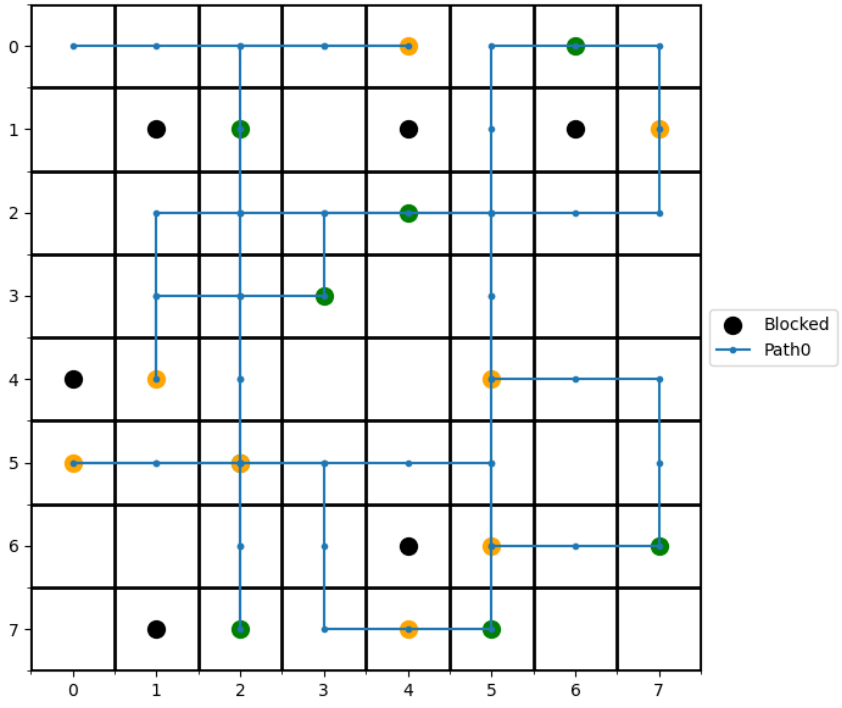

图 5-1 问题 3 _8*8 机器人路径图

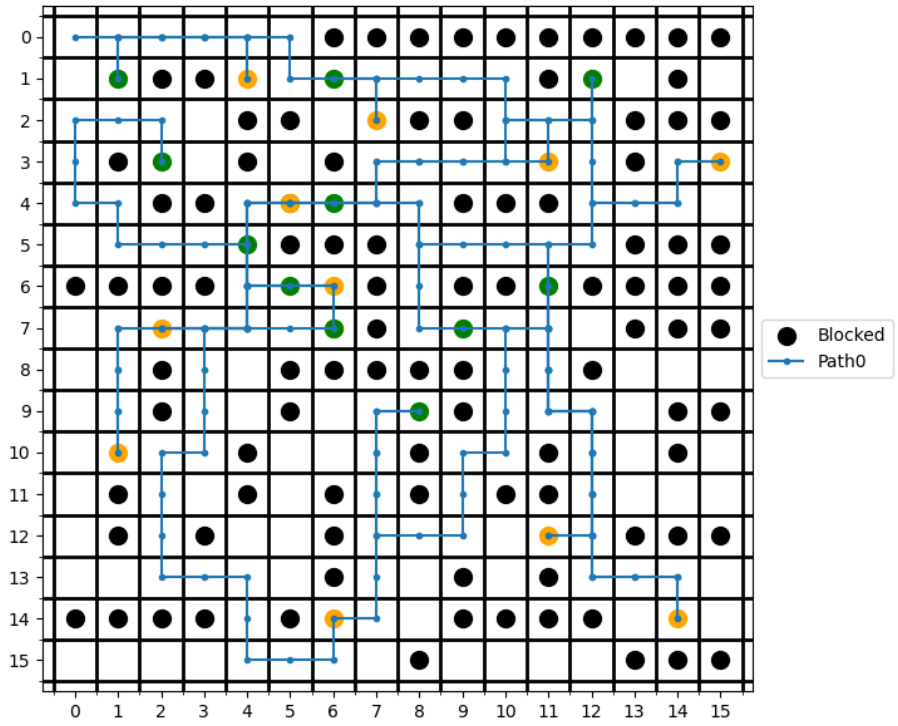

图 5-2 问题 3 _16*16 机器人路径图

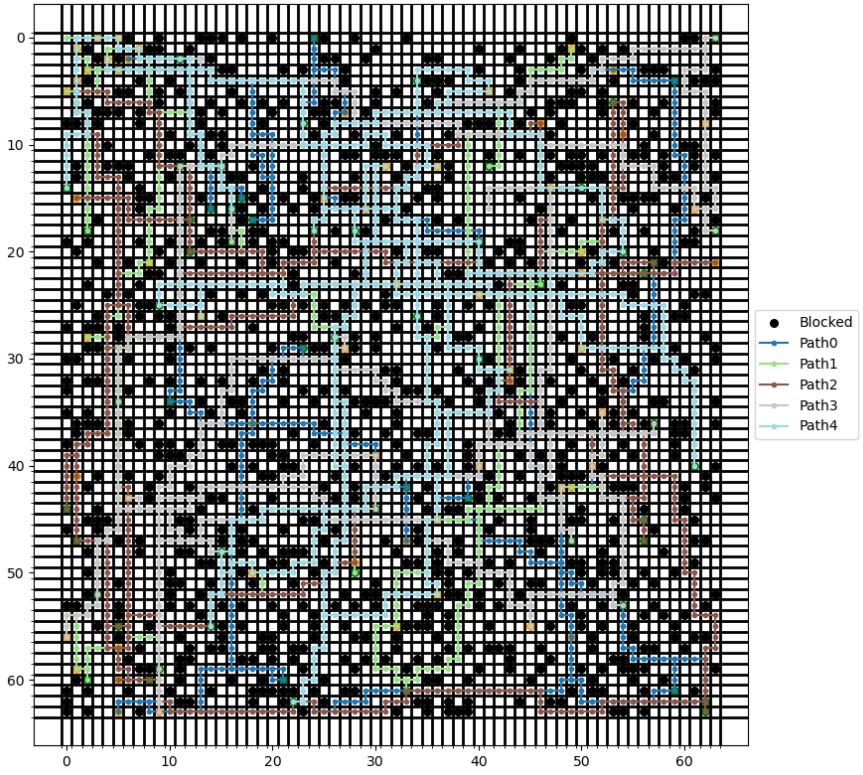

图 5-3 问题 3 _64*64 机器人路径图

六、问题四

6.1 问题四的分析

问题 4 要求在问题 3 的基础上，假设部分任务已经被指定的机器人来完成，这里任务完成的先后不做固定。模型只需沿用问题 3 模型，将部分已经指定的任务作为约束条件，并对算法进行改进求解。

6.2 数据预处理

通过处理附件 4 数据发现它是在附件三基础上加上一个机器人已安排任务，所以数据处理与大部分与第三题差异不大，只需额外读取各机器人已安排的任务即可，其中附件 4 前两个文件中的机器人任务安排见表 6-1。

表 6-1 附件 4 前两个文件任务安排

文件	8x8map	16x16map
	已分配任务编号	已分配任务编号
机器人 1	[1,2,3]	[1,2,6]
机器人 2	无	无

6.3 基于问题三的整数线性规划模型的建立

6.3.1 模型建立

新增约束条件确定

$$y_{ik} = 1; i \in R, k \in \vartheta_i \qquad (6-1)$$

其中ϑ_i表示机器人i已安排的任务集合。

其余完全与第三题模型相同。

6.3.2 整体整数线性模型

$$Min\ \tau$$

$$s.t.\begin{cases}
y_{ik}=1; i\in R, k\in \vartheta_i \\
x_{io_i1}=1; i\in R \\
\sum_{\forall t\in T} x_{ijt}=1; i\in R, j\in S, t\in T \\
x_{i_1jt}+x_{i_2jt}\le 1; i_1,i_2\in R, i_1\ne i_2, j\in S, t\in T \\
x_{i_1j_1t-1}+x_{i_1j_2t}+x_{i_2j_2t-1}+x_{i_2j_1t}\le 3; i_1,i_2\in R, i_1\ne i_2, j_1\in S, j_2\in \delta_{j_1}, t\in T\backslash\{1\} \\
x_{ijt-1}\le \sum_{\forall j'\in \delta_j\cup\{j\}} x_{ij't}; i\in R, j\in S, t\in T\backslash\{1\} \\
\sum_{\forall i\in R} y_{ik}=1; k\in K \\
b_{ikt}\le \frac{1}{2}\left(x_{iq_kt}+y_{ik}\right); i\in R, t\in T, k\in K \\
c_{ikt}\le \frac{1}{2}\left(x_{iz_kt}+y_{ik}\right); i\in R, t\in T, k\in K \\
\sum_{\forall t\in T} t\cdot b_{ikt}\le \sum_{\forall t\in T} t\cdot c_{ikt};\ i\in R, k\in K \\
\sum_{\forall i\in R}\sum_{\forall t\in T} b_{ikt}=1; k\in K \\
\sum_{\forall i\in R}\sum_{\forall t\in T} c_{ikt}=1; k\in K \\
0\le \sum_{t'=1}^{t}\sum_{\forall k\in K} b_{ikt'}-c_{ikt'}\le 1; i\in R, t\in T \\
\sum_{\forall k\in K}\sum_{\forall t\in T} b_{ikt}=\sum_{\forall k\in K} y_{ik}; i\in R \\
\sum_{\forall k\in K}\sum_{\forall t\in T} c_{ikt}=\sum_{\forall k\in K} y_{ik}; i\in R \\
x_{ijt}\le x_{ijt+1}+\left(\sum_{\forall k\in K} y_{ik}-\sum_{\forall k\in K}\sum_{t'=1}^{t} c_{ikt}\right); i\in R, j\in S, t\in T\{u\} \\
\tau\ge (t-1)\cdot\sum_{\forall k\in K} c_{ikt}; i\in R, t\in T \\
x_{ijt}\in\{0,1\}; i\in R, j\in S, t\in T \\
y_{ik}\in\{0,1\}; i\in R, k\in K \\
b_{ikt}\in\{0,1\}; i\in R, k\in K, t\in T \\
c_{ikt}\in\{0,1\}; i\in R, k\in K, t\in T \\
\tau\ge 0
\end{cases}$$

6.4 问题求解

6.4.1 基于动态规划与改进的 GCBS 算法的求解

由于上文已经发现求解器无法快速求解此模型，因此直接对第三题中算法改进以求解本题。

问题四是一个更复杂的多机器人路径规划问题，要求每个机器人只能执行特定任务集合内的任务，或者执行某些不限机器人的任务。为了适应这种更严格的任务分配约束，本文对动态规划和改进的 GCBS 算法进行了调整。

1.任务允许集的引入：在问题四中，每个机器人`k` 只能选择其允许集合`allowed[k]` 中的任务，或某些不限机器人的任务 `unrestricted_tasks`。这一设定被引入到动态规划和 GCBS 算法中，确保每个机器人仅执行其允许的任务。

2.初始状态的修改：初始状态时，每个机器人只能选择其允许集合中的任务或不受限制的任务作为第一个任务进行分配。与问题三的无约束选择不同，这一改动大大限制了机器人可选的初始任务集。

3.状态转移的条件检查：在状态转移时，算法增加了条件检查，确保只有当前机器人的允许集合中的任务或不受限制的任务才会被纳入到该机器人的任务集中。这保证了在整个任务分配和路径规划过程中，任务的选择始终满足约束条件。

6.5 结果展示

各个地图器人运输总时间和以及求解时间结果如表所示：

表 6-2 各个地图器人运输总时间和以及求解时间结果

地图	最优值	求解时间
8*8	38	24.3966
16*16	124	0.0033
64*64	781	0.8403

其中各个机器人执行任务顺序表如下表所示：

表 6-3 各个地图器人执行任务顺序表

地图	机器人编号	任务顺序
8*8	1	[1, 2, 4, 0]
	2	[7, 6, 5, 3]
16*16	1	[2, 0, 9, 5, 6, 1]
	2	[8, 3, 4, 7, 10, 11]
64*64	1	[41, 27, 3, 30, 44, 31, 36, 53, 29]
	2	[50, 6, 4, 9, 24, 17, 23, 10, 32, 7, 40, 56, 0]
	3	[39, 15, 14, 42, 25, 19, 2, 37, 21, 22, 34, 55, 57]
	4	[1, 5, 47, 48, 59, 20, 26, 16, 58, 54, 52, 51, 8]
	5	[46, 35, 33, 43, 18, 38, 12, 13, 28, 49, 11, 45]

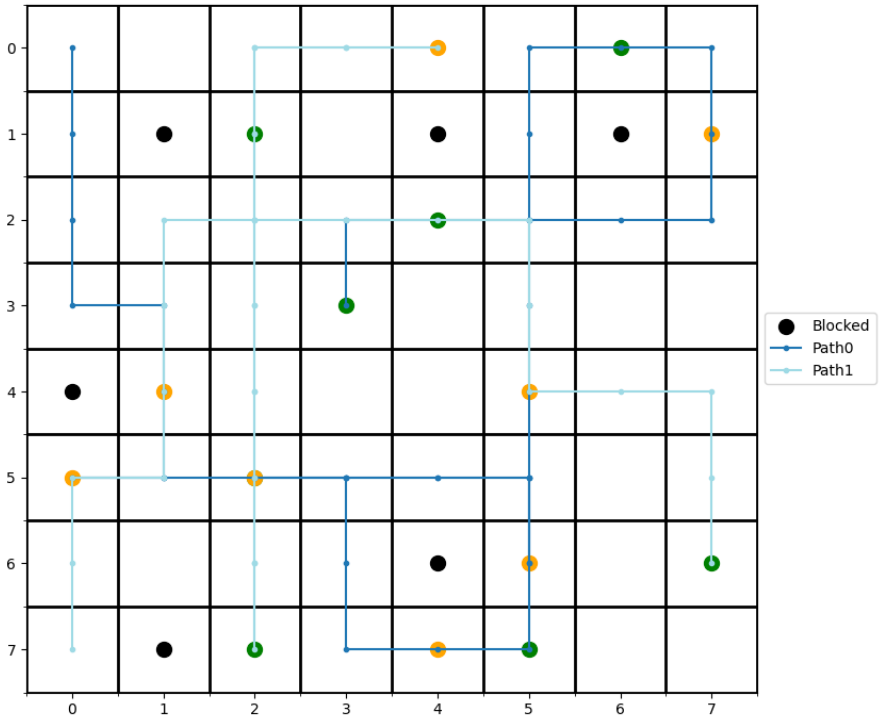

图 6-1 问题 4 _8*8 机器人路径图

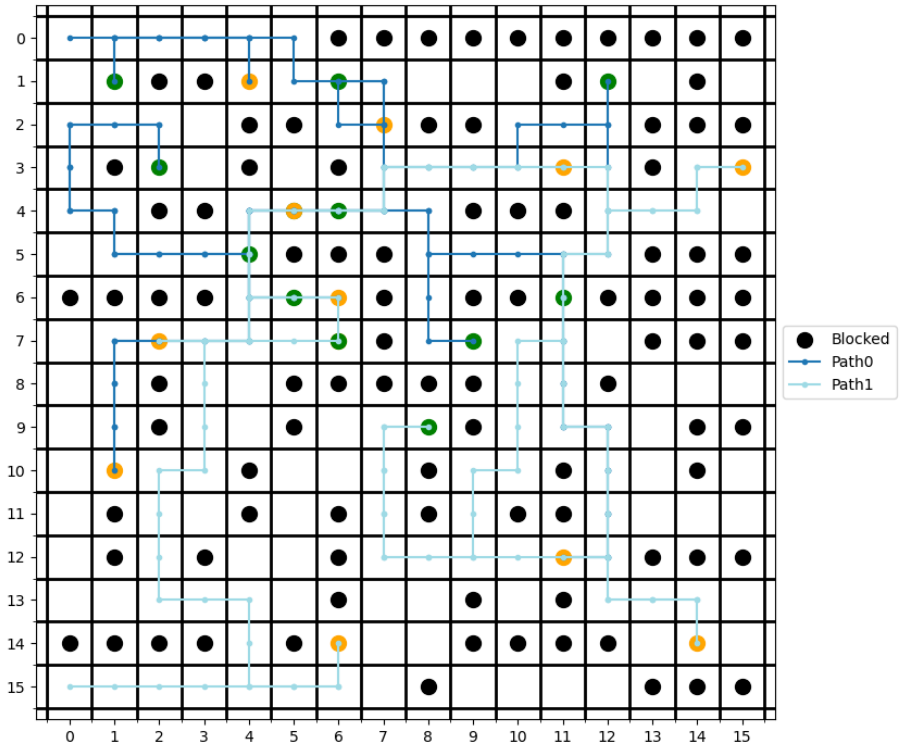

图 6-2 问题 4 _16*16 机器人路径图

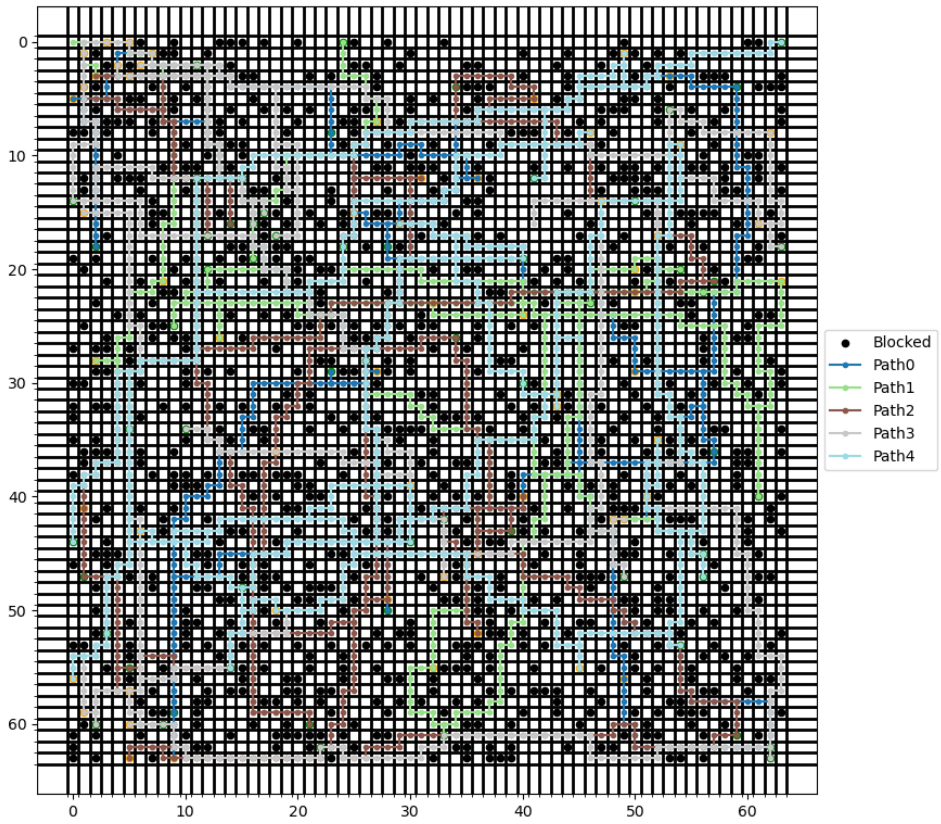

图 6-3 问题 4 _64*64 机器人路径图

七、模型的评价与改进

9.1 模型优点

1、建立了整数线性规划模型很好地描述了问题的约束与目标。

2、使用求解器对求出了小规模问题的精确解。

3、设计了改进算法在较短时间内求出了高质量的解。

9.2 模型缺点

1、整数线性规划需要将复杂约束线性化，增加模型复杂度。

2、求解器只能解决小规模问题，求解大规模问题时耗时较长。

3、GCBS 和 A*算法只能找到近似最优解无法找到全局最优解

9.3 模型推广与改进

1、进一步挖掘问题中的约束关系，建立复杂程度更低的模型

2、设计性能更好的算法以在更短的时间内获得最优解。

参考文献

[1] 王祥丰, 李文浩, 机器学习驱动的多智能体路径搜寻算法综述, 运筹学学报, 27(04):106-135, 2023.

[2] Yang L, Li P, Qian S, et al., Path Planning Technique for Mobile Robots: A Review, Machines, 11(10):980, 2023.

[3] Yu J, LaValle S M. Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics[J]. IEEE Transactions on Robotics, 2016, 32(5): 1163-1177.

附录

附录 1：部分读取数据 python 代码

```
import numpy as np
import networkx
import matplotlib.pyplot as plt
def read_data3(path):
    is_ok = True
    with open(path, 'r') as file:
        lines = file.readlines()
    map_size = lines[1].strip().split()
    map_height = int(map_size[0])
    map_width = int(map_size[1])
    map_array = []
    for i in range(2, 2 + map_height):
        row = lines[i].strip().split()
        # row = [0 if i == '.' else 1 for i in row]
        map_array.append(row)
    i += 3
    agent_pos = []
    while lines[i].strip() != 'tasks' and lines[i].strip() != 'task':
        pos = list(map(int, lines[i].strip().split()))
        agent_pos.append((pos[0], pos[1]))
        i += 1
    tasks = []
    ind = i + 1
    num_tasks = int(lines[ind].strip())
    for i in range(ind+1, ind + num_tasks+1):
        task = list(map(int, lines[i].strip().split()))
        tasks.append([(task[0], task[1]), (task[2], task[3])])
    for task in tasks:
        if map_array[task[0][0]][task[0][1]] == 1 or map_array[task[1][0]][task[1][1]] ==
1:
            print("机器人的起点或终点有误"+ str(task))
```

```
                is_ok = False
                break
    return map_array,agent_pos, tasks,is_ok
def is_valid(x, y, map_array):
    return 0 <= x < len(map_array) and 0 <= y < len(map_array[0])
def plot_graph(map_array,st):
```

附录 2：部分求解器求解方法 python 代码

```
import gurobipy as gp
from gurobipy import GRB, quicksum
# 第四题模型求解
def
Gurobi_3_4_ILP(u,st_key,o_key,pre_task,node_dict,node_dict_reversed,neighbor,timelimit
):
    # 数据
    m = len(o_key)
    n = len(node_dict)
    p = len(st_key)
    R = range(m)
    S = range(n)
    T = range(u)
    K = range(p)
    o = o_key
    q = [i[0] for i in st_key]
    z = [i[1] for i in st_key]
    try:
        model = gp.Model('Q3_4_ILP')
        x = model.addVars(R, S, T, vtype=GRB.BINARY, name='x')
        y = model.addVars(R, K, vtype=GRB.BINARY, name='y')
        b = model.addVars(R, K, T, vtype=GRB.BINARY, name='b')
        c = model.addVars(R, K, T, vtype=GRB.BINARY, name='c')
        tau = model.addVar(vtype=GRB.INTEGER, name='y')
        model.setObjective(tau, GRB.MINIMIZE)
        model.addConstrs((y[i, k] == 1 for i in R for k in pre_task[i]), name='pre_task')
        model.addConstrs((x[i, o[i], 0] == 1 for i in R), name='start')
```

```
        model.addConstrs(
            ( quicksum(x[i, j, t] for j in S) == 1
            for i in R for t in T)
            ,name = 'one_point'
        )
        if m > 1:
            model.addConstrs(
                ( x[i1,j,t] + x[i2,j,t] <= 1
                for i1 in R for i2 in R for j in S for t in T if i1 != i2)
                ,name = '2_robot_no_1point'
            )
        if m > 1:
            model.addConstrs(
                ( x[i1,j1,t-1] + x[i1,j2,t] + x[i2,j1,t] + x[i2,j2,t-1] <= 3
                for i1 in R for i2 in R for j1 in S for j2 in neighbor[j1] for t in T[1:] if
i1 != i2)
                ,name = 'change_conflict'
            )
        model.addConstrs(
            ( x[i, j, t-1] <= quicksum(x[i, j1, t] for j1 in neighbor[j] +[j] )
            for i in R for j in S for t in T[1:])
            ,name = 'neighbor'
        )
        model.addConstrs(
            ( quicksum(y[i, k] for i in R) == 1
            for k in K)
            ,name = 'one_task_per1robot'
        )
        model.addConstrs(
            ( b[i, k, t] <= 0.5*(y[i, k] + x[i, q[k], t])
            for i in R for k in K for t in T)
            ,name = 'pick_up'
        )
        model.addConstrs(
```

```
        ( c[i, k, t] <= 0.5*(y[i, k] + x[i, z[k], t])
        for i in R for k in K for t in T)
        ,name = 'deliver'
    )
    model.addConstrs(
        ( quicksum(b[i, k, t] for t in T for i in R) == 1
        for k in K)
        ,name = 'pick_up_once'
    )
    model.addConstrs(
        ( quicksum(c[i, k, t] for t in T for i in R) == 1
        for k in K)
        ,name = 'deliver_once'
    )
    model.addConstrs(
        ( quicksum(t*b[i, k, t] for t in T) <= quicksum(t*c[i, k, t] for t in T)
        for i in R for k in K)
        ,name = 'pick_up_earlier'
    )
    model.addConstrs(
        ( quicksum(b[i, k, t1] -c[i, k, t1] for k in K for t1 in range(t)) <= 1
        for i in R for t in T)
        ,name = 'robot_nomore1'
    )
    model.addConstrs(
        ( quicksum(b[i, k, t1] -c[i, k, t1] for k in K for t1 in range(t)) >= 0
        for i in R for t in T)
        ,name = 'robot_up0'
    )
    model.addConstrs(
        ( quicksum(b[i, k, t] for t in T for k in K) == quicksum(y[i, k] for k in K)
        for i in R)
        ,name = 'pick_up_deliver'
    )
```

```
            model.addConstrs(
                ( quicksum(c[i, k, t] for t in T for k in K) == quicksum(y[i, k] for k in K)
                for i in R)
                ,name = 'send_deliver'
            )
            model.addConstrs(
                ( x[i,j,t] <= x[i,j,t+1] + (quicksum(y[i, k] for k in K) -quicksum(c[i, k, t1] for
k in K for t1 in range(t)))
                for i in R for j in S for t in T[:-1])
                ,name = 'stop'
            )
            model.addConstrs(
                ( tau >= t * quicksum(c[i, k, t] for k in K)
                for i in R for t in T)
                ,name = 'complete_time'
            )
            model.setParam('outputflag', 0)
            model.setParam('TimeLimit', timelimit)
            model.setParam('threads', 12)
            # 求解
            model.optimize()
            # 输出结果
            if model.SolCount >0:
                opt_x = [[[x[i, j, t].x for t in T] for j in S] for i in R]
                opt_y = [[y[i, k].x for k in K] for i in R]
                opt_b = [[[b[i, k, t].x for t in T] for k in K] for i in R]
                opt_c = [[[c[i, k, t].x for t in T] for k in K] for i in R]
                opt_tau = model.objVal
                status = model.status
                gap = model.MIPGap
                time = model.Runtime
            return opt_x, opt_y,opt_b,opt_c,opt_tau, status, gap, time
        except gp.GurobiError as e:
            print('Error code ' + str(e.errno) + ": " + str(e))
```

```
from Q3_readdata import read_data1,plot_graph,read_data3,read_data4_1
timelimit = 100
if __name__ == '__main__':
    us = [81,197,4771]
    # 规定任务分配
    # 8x8：1 7--> 3 3--> 5 0--> 7 5--> 7 4--> 5 2
    # 16x16: 2 7--> 1 6--> 10 1--> 7 9--> 6 6--> 1 12
    pre_task_assign = [
                        [[(1,7),(3,3)],[(5,0),(7,5)],[(7,4),(5,2)]],
                        [[(2,7),(1,6)],[(10,1),(7,9)],[(6,6),(1,12)]]
                        ]
    pre_task = [[[0,1,2],[]],[[0,1,5],[]]]
    folder = 'D:/git 仓库/mathmodel_2024/questions/集训模型 3（研）/附件 4/'
    file_names = ['8x8map.txt', '16x16map.txt', '64x64map.txt']
    map_array,agent_pos,st,is_ok = read_data4_1(folder + file_names[1])
    G,node_dict,st_key,node_dict_reversed,neighbor = plot_graph(map_array,st)
    o_key = [node_dict_reversed[i] for i in agent_pos]
    m = len(o_key)
    n = len(node_dict)
    p = len(st_key)
    u = 81
    z = [i[1] for i in st_key] # 终点
    print('开始求解')
    opt_x, opt_y,opt_b,opt_c,opt_tau, status, gap, time = Gurobi_3_4_ILP(u,st_key,o_key,pre_task[1],node_dict,node_dict_reversed,neighbor,timelimit)
    project = [[] for i in range(m)]
    for i in range(m):
        for t in range((int)(opt_tau+1)):
            for j in range(n):
                if opt_x[i][j][t] == 1:
                    project[i].append(node_dict[j])
    task_prijject = [[] for i in range(m)]
    for i in range(m):
```

```
            for k in range(p):
                for t in range(u):
                    if opt_c[i][k][t] == 1:
                        task_priject[i].append(k)
    print('任务分配')
    for i in task_priject:
        print('机器人',task_priject.index(i),':',i)
    print('各机器人路径：')
    for i in project:
        print(i)
    print('最优值：',opt_tau)
    print('求解状态：',status)
    print('gap:',gap)
    print('求解时间：',time)
```

附录三：部分 A*算法和 GCBS 算法代码

```
import heapq
# 定义方向：上、下、左、右
DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1), (0, 0)]  # 可以原地等待
# 计算曼哈顿距离作为启发式函数
def manhattan_distance(point1, point2):
    return abs(point1[0] - point2[0]) + abs(point1[1] - point2[1])
# 检查节点是否在网格范围内并且可通行
def is_valid_position(grid, position):
    x, y = position
    return 0 <= x < len(grid) and 0 <= y < len(grid[0]) and grid[x][y] != '@'
# A*算法实现
def a_star(grid, start, goal, constraints):
    open_list = []
    closed_list = set()
    start_node = (
        0 + manhattan_distance(start, goal), 0, manhattan_distance(start, goal), start,
        None)  # (f, g, h, position, parent)
    heapq.heappush(open_list, start_node)
```

```
    while open_list:  # 当 open_list 不为空
        current_node = heapq.heappop(open_list)  # 将 f 值最小的点选出并从 open_list 中去除
        f, g, h, current_position, parent = current_node
        # 到达终点
        if current_position == goal:
            path = []
            while current_node:
                path.append(current_node[3])
                current_node = current_node[4]
            return path[::-1]  # 返回从起点到终点的路径
        if current_position in closed_list:
            continue
        for direction in DIRECTIONS:
            neighbor_position = (current_position[0] + direction[0], current_position[1] + direction[1])
            if neighbor_position in closed_list:
                continue
            if not is_valid_position(grid, neighbor_position):
                continue
            if (g + 1, neighbor_position) in constraints:  # 点冲突约束
                continue
            if (g + 1, (current_position, neighbor_position)) in constraints:  # 边冲突约束
                continue
            new_g = g + 1
            new_h = manhattan_distance(neighbor_position, goal)
            new_f = new_h + new_g
            i_list = [i for i, node in enumerate(open_list) if neighbor_position == node[3]]
            if i_list:
                i = i_list[0]  # 正常情况下 i_list 只有一个元素或者为空
                if new_g < open_list[i][1]:  # 比较 g 值看是否更新节点
                    open_list.pop(i)  # 确定更新后删除原节点信息
                    neighbor_node = (new_f, new_g, new_h, neighbor_position,
```

```
current_node)
                        heapq.heappush(open_list, neighbor_node)
                    else:
                        continue
                else:
                    neighbor_node = (new_f, new_g, new_h, neighbor_position,
current_node)
                    heapq.heappush(open_list, neighbor_node)
            closed_list.add(current_position)
        return None  # 没有找到路径
import heapq
from itertools import count
import matplotlib.pyplot as plt
from adjustText import adjust_text
import preprocess
import route_graph
import pandas as pd
import time
import A_star


# 检查是否存在顶点或边冲突
def detect_conflicts(paths):
    conflicts = []
    max_time = max(len(path) for path in paths)
    for t in range(max_time):  # 点冲突检测
        occupied_positions = {}
        for i, path in enumerate(paths):
            if t <= len(path) - 1:  # 到终点后下一个时间步长机器消失，既不检查点
冲突
                pos = path[t]
                if pos in occupied_positions:
                    conflicts.append({'type': 'vertex', 'step': t, 'robots': [i,
occupied_positions[pos]]})
```

```
                    occupied_positions[pos] = i

        for i, path_i in enumerate(paths):  # 边冲突检测
            if t < len(path_i) - 1:
                for j, path_j in enumerate(paths):
                    if j <= i:
                        continue
                    if t < len(path_j) - 1:
                        if path_i[t] == path_j[t + 1] and path_i[t + 1] == path_j[t]:
                            conflicts.append({'type': 'edge', 'step': t + 1, 'robots': [i,
j]})
    if conflicts:
        return conflicts[0], len(conflicts)
    else:
        return None, 0

# CBS 主算法
def cbs(grid, starts_goals):
    open_list = []
    counter = count()  # 用于生成唯一的计数器
    paths = [A_star.a_star(grid, start_goal[0], start_goal[1], {}) for start_goal in
starts_goals]
    root = {
        'constraints': {},
        'paths': paths,
        'cost': max(len(path) - 1 for path in paths)
    }
    heapq.heappush(open_list, (root['cost'], next(counter), root))

    while open_list:
        _, _, node = heapq.heappop(open_list)

        # 检查冲突
        conflict, conflict_num = detect_conflicts(node['paths'])
```

```
            if not conflict:
                return node['paths']

            # 生成两个子问题, 一个是第一个机器人让步，另一个是第二个机器人让步
            robots = conflict['robots']
            step = conflict['step']

            for robot in robots:
                new_constraints = {**node['constraints']}
                if conflict['type'] == 'vertex':
                    new_constraints.setdefault(robot, []).append((step, node['paths'][robot][step]))  # 如果有冲突则
                elif conflict['type'] == 'edge':
                    new_constraints.setdefault(robot, []).append(
                        (step, (node['paths'][robot][step - 1], node['paths'][robot][step])))

                new_paths = node['paths'][:]
                new_paths[robot] = A_star.a_star(grid, starts_goals[robot][0], starts_goals[robot][1],
                                                 new_constraints.get(robot))  # 添加冲突约束重新计算
                if new_paths[robot]:
                    new_node = {
                        'constraints': new_constraints,
                        'paths': new_paths,
                        'cost': max(len(path) - 1 for path in new_paths) + conflict_num
                    }
                    heapq.heappush(open_list, (new_node['cost'], next(counter), new_node))

        return None  # 没有找到解

# 测试用例
if __name__ == "__main__":
```

```
    folder = 'C:/Users/25492/Desktop/集训模型 3（研）/附件 1/'
    file_names = ['8x8map.txt', '16x16map.txt', '64x64map.txt']
    grid, st, ok = preprocess.read_data1(folder + file_names[2])
    if ok:
        time1 = time.time()
        paths = cbs(grid, st)
        time2 = time.time()
        solution_time = time2 - time1
        print(f'计算时间：{solution_time}')
        if paths:
            route_graph.plot_map(grid, paths, st)
            for i, path in enumerate(paths):
                print(f"机器人 {i} 的路径: {path}")
            plt.show()
            time_cost = max([len(path) - 1 for path in paths])
            print(f'时间开销：{time_cost}')
        else:
            print("未找到可行路径")
```

附录 4：部分动态规划与改进 GCBS 算法代码

```
import time
import numpy as np
import problem2 as p2
import preprocess



def Calculate_T(cost, assign_roles, remain_roles, d, d_s):
    T = np.zeros((len(cost), len(remain_roles)))
    for i in range(len(cost)):
        for j, role_j in enumerate(remain_roles):
            if not assign_roles[i]:
                T[i, j] = d_s[i, role_j]
            else:
                T[i, j] = d[assign_roles[i][-1], role_j] + cost[i]  # path 本身长度加上末尾任务终点到该任务起点的步长
```

```
    return T


def greedy_main(d, d_s):
    m = d_s.shape[0]  # 机器人数量
    n = d_s.shape[1]  # 任务数
    cost = [0]*m # agent 的目前行驶距离
    assign_roles = [[] for _ in range(m)]
    remain_roles = list(range(n))
    while remain_roles:
        T = Calculate_T(cost, assign_roles, remain_roles,d,d_s)
        flat_index = np.argmin(T)
        i, j = np.unravel_index(flat_index, T.shape)
        role = remain_roles[j]
        assign_roles[i].append(role)
        cost[i] = T[i, j] + d[role, role]
        remain_roles.remove(role)
    return assign_roles,cost
if __name__ == '__main__':
    file_names1 = ['d3_8x8.npy', 'd3_16x16.npy', 'd3_64x64.npy']
    file_names2 = ['d_start3_8x8.npy', 'd_start3_16x16.npy', 'd_start3_64x64.npy']
    k = 2
    dist = np.load(file_names1[k])
    dist_start = np.load(file_names2[k])
    n = dist_start.shape[0]  # 机器人数量
    m = dist_start.shape[1]  # 任务数
    time1 = time.time()
    assignment,cost = greedy_main(dist, dist_start)
    time2 = time.time()
    solution_time = time2 - time1
    print('计算时间：', solution_time)
    print("最小总时间:", max(cost))
    for i in range(n):
        print(f"机器人 {i} 的任务顺序:", assignment[i])
import heapq
```

```
from itertools import count
import matplotlib.pyplot as plt
import time
import preprocess
import route_graph
import A_star
import Greedy3
import numpy as np
import DP3_12
# 检查是否存在顶点或边冲突
def detect_conflicts(paths):
    max_time = max(len(path) for path in paths)
    conflicts = []
    for t in range(max_time):  # 点冲突检测
        occupied_positions = {}
        for i, path in enumerate(paths):
            step = min(t, len(path) - 1)
            pos = path[step]
            if pos in occupied_positions:
                conflicts.append({'type': 'vertex', 'step': [step,
occupied_positions[pos][1]],
                                  'robots': [i, occupied_positions[pos][0]]})
            occupied_positions[pos] = [i, step]

        for i, path_i in enumerate(paths):  # 边冲突检测
            if t < len(path_i) - 1:
                for j, path_j in enumerate(paths):
                    if j <= i:
                        continue
                    if t < len(path_j) - 1:
                        if path_i[t] == path_j[t + 1] and path_i[t + 1] == path_j[t]:
                            conflicts.append({'type': 'edge', 'step': [t + 1, t + 1],
'robots': [i, j]})
        if conflicts:
```

```
                return conflicts[0], len(conflicts)
            else:
                return None, 0
def calculate_ori_path(robot, grid, agent_pos, st, roles):
    path_start1 = A_star.a_star(grid, agent_pos[robot], st[roles[robot][0]][0], {})
    path_start2 = A_star.a_star(grid, st[roles[robot][0]][0], st[roles[robot][0]][1], {})[1:]
    path = path_start1 + path_start2
    length = len(path_start1) + len(path_start2)
    len_list = [len(path_start1), length]
    for role in roles[robot][1:]:
        gap_path = A_star.a_star(grid, path[-1], st[role][0], {})[1:]
        role_path = A_star.a_star(grid, st[role][0], st[role][1], {})[1:]
        length += len(gap_path)
        len_list.append(length)
        length += len(role_path)
        len_list.append(length)
        path += gap_path
        path += role_path
    return path, len_list
def calculate_constraint_path(robot, grid, agent_pos, st, roles, constraints):
    ori_path, len_list = calculate_ori_path(robot, grid, agent_pos, st, roles)
    if constraints:
        constraints_split = [[] for _ in range(len(len_list))]
        for c in constraints:
            for i in range(1, len(len_list)):
                if len_list[i] > c[0] >= len_list[i - 1]:
                    c[0] = c[0] + 1 - len_list[i - 1]
                    constraints_split[i].append(c)
        path_start1 = A_star.a_star(grid, agent_pos[robot], st[roles[robot][0]][0],
constraints_split[0])
        path_start2 = A_star.a_star(grid, st[roles[robot][0]][0], st[roles[robot][0]][1],
constraints_split[1])[1:]
        path = path_start1 + path_start2
        for i, role in zip(range(0, len(constraints_split), 2), roles[robot]):
```

```
                if i == 0:
                    path_start1 = A_star.a_star(grid, agent_pos[robot], st[roles[robot][0]][0], constraints_split[i])
                    path_start2 = A_star.a_star(grid, st[roles[robot][0]][0], st[roles[robot][0]][1],
                                                constraints_split[i + 1])[1:]
                    path = path_start1 + path_start2
                else:
                    gap_path = A_star.a_star(grid, path[-1], st[role][0], constraints_split[i])[1:]
                    role_path = A_star.a_star(grid, st[role][0], st[role][1], constraints_split[i + 1])[1:]
                    path += gap_path
                    path += role_path
            return path
        else:
            return ori_path


# CBS 主算法
def cbs(grid, agent_pos, st, roles):
    open_list = []
    counter = count()  # 用于生成唯一的计数器
    paths = [calculate_ori_path(i, grid, agent_pos, st, roles)[0] for i in range(len(agent_pos))]
    root = {
        'constraints': {},
        'paths': paths,
        'cost': max(len(path) - 1 for path in paths)
    }
    heapq.heappush(open_list, (root['cost'], next(counter), root))

    while open_list:
        _, _, node = heapq.heappop(open_list)
```

```
            # 检查冲突
            conflict, conflict_num = detect_conflicts(node['paths'])
            if not conflict:
                return node['paths']

            # 生成两个子问题
            robots = conflict['robots']
            steps = conflict['step']

            for robot, step in zip(robots, steps):
                new_constraints = {**node['constraints']}
                if conflict['type'] == 'vertex':
                    new_constraints.setdefault(robot,                    []).append((step,
node['paths'][robot][step]))   # 如果有冲突则
                elif conflict['type'] == 'edge':
                    new_constraints.setdefault(robot, []).append(
                        (step, (node['paths'][robot][step - 1], node['paths'][robot][step])))

                new_paths = node['paths'][:]
                new_paths[robot]  =  calculate_constraint_path(robot,  grid,  agent_pos,  st,
roles,

new_constraints.get(robot))   # 添加冲突约束重新计算
                if new_paths[robot]:
                    new_node = {
                        'constraints': new_constraints,
                        'paths': new_paths,
                        'cost': max(len(path) - 1 for path in new_paths) + conflict_num
                    }
                    heapq.heappush(open_list,        (new_node['cost'],        next(counter),
new_node))

        return None   # 没有找到解
```

```
# 测试用例
if __name__ == "__main__":
    k = 0
    folder = 'C:/Users/25492/Desktop/集训模型 3（研）/附件 3/'
    file_names = ['8x8map.txt', '16x16map.txt', '64x64map.txt']
    grid, agent_pos, st, ok = preprocess.read_data3(folder + file_names[k])
    file_names1 = ['d3_8x8.npy', 'd3_16x16.npy', 'd3_64x64.npy']
    file_names2 = ['d_start3_8x8.npy', 'd_start3_16x16.npy', 'd_start3_64x64.npy']
    dist = np.load(file_names1[k])
    dist_start = np.load(file_names2[k])

    if ok:
        time1 = time.time()
        assignment_roles, _ = Greedy3.greedy_main(dist, dist_start)  # 贪心算法求解任务分配
        #_, assignment_roles = DP3_12.tsp_single_robot(dist, dist_start)  # 动态规划求解任务分配
        #assignment_roles = [[1, 11, 7, 9, 4, 8, 3, 5, 10, 6, 0, 2]]
        paths = cbs(grid, agent_pos, st, assignment_roles)  # cbs 计算
        time2 = time.time()
        solution_time = time2 - time1
        print(f'计算时间：{solution_time}')
        for i, role in enumerate(assignment_roles):
            print(f"机器人 {i} 的任务顺序:", role)
        if paths:
            route_graph.plot_map(grid, paths, st)
            for i, path in enumerate(paths):
                print(f"机器人 {i} 的路径: {path}")
            plt.show()
            time_cost = max([len(path) - 1 for path in paths])
            print(f'最小总时间：{time_cost}')
        else:
            print("未找到可行路径")
```