基于 A* 算法与 CBS 算法的多机器人路径规划优化模型研究

摘　要

在现代供应链管理中，自动化仓储系统的智能搬运问题成为关键研究领域。本文针对仓储系统中多机器人路径规划与调度问题，通过建立基于多机器人路径规划的整数线性规划模型、A* 与 CBS 算法、遗传算法、混合整数线性规划模型等模型解决了以机器人运输总时间最小化为目标的路线方案等问题。

针对问题一，在确保机器人之间不发生碰撞的前提下，最小化所有机器人从起点到指定终点并完成任务所需的总时间。为解决这一问题，我们构建了**基于多机器人路径规划的整数线性规划模型**，以机器人运输时间最小化为目标函数，并加上路径约束、移动约束等约束条件，采用 **A* 算法**和 **CBS 算法**相结合的方法进行求解。A* 算法用于单机器人路径的最优规划，而 CBS 算法通过分层处理机器人之间的路径冲突，从而优化多机器人系统的整体路径，进而求解了在不同规模的地图上机器人总运输时间最小化：8*8 的地图为 **46 单位时间**，16*16 的地图为 **136 单位时间**，64*64 的地图为 **2227 单位时间**。同时，问题一的 A* 算法时间复杂度为 $\mathcal{O}(V^n)$，空间复杂度为 $\mathcal{O}(V^n)$，CBS 算法复杂度为 $\mathcal{O}(n \cdot \mathcal{O}(A^*))$，空间复杂度为 $\mathcal{O}(n \cdot V)$。

针对问题二，在问题一的基础上假设每个机器人在完成任务后停留在终点而不消失，为了解决这一问题，我们重新建立**整数线性规划模型**，以机器人运输时间最小化为目标函数，并引入机器人完成任务后停留在目的地的约束条件，采用了 A* 算法和 CBS 算法相结合的方法进行求解，求解了在不同规模的地图上机器人总运输时间最小化：8*8 的地图为 **46 单位时间**，16*16 的地图为 **150 单位时间**，64*64 的地图为 **2229 单位时间**。同时，问题二中的算法复杂度与问题一中一致。

针对问题三，假设机器人数量与任务数量不相等，为了确保机器人之间不发生冲突，我们建立了**基于多机器人路径规划的混合整数线性规划模型**，以机器人运输时间最小化为目标函数。此外，我们结合**遗传算法**建立了**基于遗传算法的混合路径规划策略**，用遗传算法确定任务调度顺序通过 A* 算法找两个点间的最优路径，若机器人数量不唯一，使用 CBS 算法消除路径冲突。最终，我们求得在不同规模的地图上机器人总运输时间最小化：8*8 的地图为 **65 单位时间**，16*16 的地图为 **201 单位时间**，64*64 的地图为 **1051 单位时间**。同时，问题三的遗传算法时间复杂度为 $\mathcal{O}(T \times N \times (n \times A + L))$，空间复杂度为 $\mathcal{O}(L \times N)$。

针对问题四，假设部分任务已被固定分配给特定的机器人，我们重新构建**混合整数规划模型**，以机器人运输时间最小化为目标函数。此外，我们结合遗传算法建立了基于遗传算法的混合路径规划策略，求得在不同规模的地图上机器人总运输时间最小化：8*8 的地图为 **46 单位时间**，16*16 的地图为 **136 单位时间**，64*64 的地图为 **1168 单位时间**。同时，问题三的遗传算法算法复杂度与问题四中一致。

关键词:　整数线性规划模型 **A*** 算法 遗传算法 **CBS** 算法

一　问题的背景和重述

1.1　问题背景

在现代供应链管理中，自动化仓储系统的应用日益广泛，特别是在第三方物流（3PL）企业、鞋服行业、电商平台以及 3C 制造领域。随着业务模式的多样化和市场发展的迅速，传统仓储系统在应对快速物流需求时显得力不从心，面临存储密度低、人工效率不足等问题。因此，开发一种能够快速定位和运输货物的自动化仓储系统成为解决这些问题的关键。在这个背景下，自动化仓储系统中的智能搬运问题逐渐成为研究热点。通过智能机器人完成货物搬运任务，可以大幅提高物流效率，减少人工操作中的错误和成本。然而，如何在保证高效性的同时避免机器人之间的碰撞，并确保最小化运输总时间，仍然是一个需要深入研究的课题。因此，针对这一问题，建立数学模型和设计调度算法，对提升现代仓储系统的智能化水平具有重要意义。

1.2　问题重述

未了在不同的设定下确保运输总时间最小化并避免机器人之间的冲突。本文根据附件的数据建立数学模型，解决以下几个关键问题：

1. 多个机器人需要从各自的起点移动到指定的终点，并在完成任务后消失。每个机器人在每个时间步只能选择移动到相邻网格或原地等待。我们需要建立一个数学模型，设计出最优的路径规划方案，使得所有机器人的总运输时间最小化，并避免发生碰撞。

2. 在问题 1 的基础上，假设每个机器人在完成任务后停留在终点而不消失。我们需要修改原有的模型，重新设计路径规划方案，确保在不发生碰撞的情况下，所有机器人总运输时间最小化。

3. 在问题 2 的基础上，假设机器人数量与任务总数不相等。我们需要为每个机器人分配任务，建立新的数学模型，在避免碰撞的前提下，实现所有任务的最优调度，使总运输时间最小化。

4. 在问题 3 的基础上，进一步假设部分任务已被固定分配给特定的机器人完成。我们需要建立一个更复杂的模型和调度算法，确保在不发生碰撞的前提下，所有机器人以最优顺序完成任务，并最小化总运输时间。

二　问题分析

2.1　问题一的分析

问题一的核心是多机器人路径规划优化问题，目标是在确保机器人之间不发生碰撞的前提下，最小化所有机器人从起点到指定终点并完成任务所需的总时间。该问题属于典型的多智能体路径规划（MAPF）问题。为解决这一问题，我们构建了基于多机器人路径规划的整数线性规划模型，并采用 A* 算法和冲突分解搜索算法（CBS）相结合的方法进行求解。A 算法用于单机器人路径的最优规划，而 CBS 算法通过分层处理机器人之间的路径冲突，从而优化多机器人系统的整体路径，进而求解了在不同规模的地图（如 8×8、16×16 和 64×64）上机器人总运输时间最小化。

2.2　问题二的分析

在问题二中，我们在问题一的基础上假设每个机器人在完成任务后停留在终点而不消失，这一变化增加了路径规划的复杂性。机器人在任务完成后停留在目标点上，使得这些位置变为新的障碍物，从而限制了其他机器人的移动路径。此时，路径规划不仅需要考虑机器人在移动过程中的路径冲突，还必须确保在机器人完成任务后，其停留在终点的位置不会与其他机器人的路径产生冲突。因此，问题的核心

在于如何在不发生路径冲突的前提下，优化每个机器人的路径规划，使得所有机器人完成任务所需的总时间最小化。为了解决这一问题，我们需要重新设计整数线性规划模型，并引入新的约束条件，以应对因终点驻留而产生的路径冲突风险。

2.3 问题三的分析

在问题三中，假设机器人数量与任务数量不相等，这使得问题的复杂性进一步增加。由于机器人数量不足或任务量超出，使得必须对任务进行合理的分配和调度，同时确保机器人之间不发生冲突。此时，问题的核心在于如何在任务分配与路径规划过程中，实现各机器人任务的最优调度，确保所有任务能够在最短时间内完成，同时避免路径冲突。为了应对这个挑战，模型不仅需要对路径规划进行优化，还需要引入混合整数线性规划模型来合理分配任务，并结合算法如遗传算法、A* 算法和 CBS 算法，确保整体系统在复杂环境下的高效运行。

2.4 问题四的分析

在问题四中，假设部分任务已被固定分配给特定的机器人，这进一步增加了调度和路径规划的复杂性。此时，不仅要确保机器人完成分配给它们的任务，还要保证在任务分配的基础上，机器人之间的路径冲突最小化。这一问题的核心在于如何在任务预先分配的约束条件下，优化整体的路径规划方案，确保每个机器人在不发生冲突的前提下，以最优顺序完成其分配的任务。此外，模型还需要兼顾任务固定分配的情况，重新构建混合整数规划模型，并结路径规划算法，确保所有任务的总运输时间最小化，实现系统的高效调度。

三 模型假设

1. 假设机器人执行任务的环境是静态的，即障碍物和路径不随时间变化。所有机器人在任务执行过程中，环境条件均保持不变。

2. 假设每个任务彼此独立，机器人在执行任务时不会因其他任务的状态或完成情况受到影响。这简化了任务分配和路径规划的复杂性。

3. 假设机器人只能在网格上按上下左右四个方向移动，每次移动一个单位距离，且不会发生加速或减速的情况。

4. 假设机器人在运动过程中能完全避免与其他机器人或障碍物发生碰撞，并且所有机器人在同一时间步不能占据相同的网格单元。

四　符号说明

符号	含义
R	机器人集合，总数为 n
M	任务集合，总数为 m
$s(k)$	任务 M_k 的出发点
$t(k)$	任务 M_k 的目标点
x_{ik}	二元变量，表示机器人 R_i 是否执行任务 M_k，若 $x_{ik}=1$，则任务 M_k 由机器人 R_i 执行
π_i	机器人 R_i 的所有动作序列
π_{ik}	机器人 R_i 执行任务 M_k 时的路径，即动作序列（包括机器人接受该任务时从当前地点到达任务 M_k 出发点的动作序列）
$\lvert\pi_{ik}\rvert$	机器人 R_i 执行任务 M_k 所用的时间步数
O	不可通行的顶点集合

五　模型的建立与求解

5.1　问题一模型的建立与求解

5.1.1　基于多机器人路径规划的整数线性规划模型

对于问题一假设每个机器人只有一个任务目标且在完成任务后消失，每个单位时间内机器人只能在停在原地和移动中选择一项，在不发生冲突的前提下使机器人运输总时间最小化，我们参考了多智能体路径搜索算法 [1], 发现该问题是一个典型的 MAPF 问题。MAPF 问题的核心在于多个智能体（如机器人）需要在同一个环境中从起点移动到各自的目标点，并且在这一过程中避免与其他智能体发生碰撞。因此，我们建立了基于多机器人路径规划的整数线性规划模型对运输总时间最小化进行了求解，具体模型如下：

1. 任务设置

假设每个机器人只有一个任务目标，并在完成任务后消失。对于每个机器人，定义输入为一个三元组 $\langle G, s, t\rangle$，其中：

- $G=(V,E)$：表示一个无向二维图，V 是顶点集，E 是边集。顶点集 V 中的每个元素代表一个网格单元，边集 E 表示相邻网格单元之间的连接。
- $O \in V$：表示障碍物的集合，即不可通行的顶点集合。
- $E = V - O$：表示可通行的顶点集合。
- $R=\{1,2,\ldots,i\}$：表示机器人集合。

映射 $s:[1,\ldots,k]\to V$ 将每个机器人映射到一个顶点，描述每个机器人的起始节点。映射 $t:[1,\ldots,k]\to V$ 将每个机器人映射到一个顶点，描述每个机器人的目标节点。

每个机器人在每一步的动作定义为一个函数 $a : V \to V$，即 $a(v) = v'$，表示机器人位于结点 v 并执行了动作 a，在下一个时间步将从顶点 v 移动到顶点 v'。

2. 设定动作类型

我们设定机器人在雨后的过程中有两种动作类型，具体如下：

1. **等待**：机器人将在当前节点停留一个时间步。

2. **移动**：机器人从当前节点 v 移动到图中的相邻节点 v'（仅前后左右四个方向），即 $(v, v') \in F$。

3. 路径表示

设 $\pi = (a_1, \ldots, a_n)$ 为一串动作序列，对于一个机器人 i，用 $\pi_i[x]$ 表示从源点 $s(i)$ 开始，执行前 x 个动作后的位置, 即

$$\pi_i[x] = a_x\left(a_{x-1}\left(\cdots a_1(s(i))\right)\right). \tag{1}$$

如果机器人 i 从起点 $s(i)$ 出发，完整地执行完 π 后，到达目标点 $t(i)$，即

$$\pi_i\left[\|\pi\|\right] = t(i). \tag{2}$$

那么 π_i 为机器人 i 的规划路径。

4. 目标函数

为了最小化每个智能体达到目标所需的时间步数之和，建立如下目标函数：

$$\text{minimize} \quad \sum_{1<i<k} |\pi_i| \tag{3}$$

其中，i 表示机器人，π_i 达到目标的时间步数。

5. 约束条件

a. 路径约束

每个机器人 i 的路径必须有效:

$$\pi_i(t) \in F, \quad \forall i, \forall t \tag{4}$$

其中，$\pi_i(t)$ 表示机器人 i 在时间步 t 时的位置，F 为可通行的顶点集合。

b. 移动约束

每个机器人每个时间步仅能移动到相邻的结点 v'，即仅能向前、后、左、右四个方向移动:

$$\pi_i(t+1) \in \{\pi_i(t) \cup a(\pi_i(t))\}, \quad \forall i, \forall t \tag{5}$$

其中，$a(\pi_i(t))$ 表示机器人 i 从顶点 $\pi_i(t)$ 移动到相邻顶点 v'。

c. 避免顶点冲突

在任何时间步 t 中，各机器人不能同时位于同一个顶点:

$$\pi_i(t) \neq \pi_j(t), \quad \forall i \neq j, \forall t \tag{6}$$

d. 避免边冲突

在任何时间步之间的移动中，各机器人不能同时经过同一条边:

$$(\pi_i(t), \pi_i(t+1)) \neq (\pi_j(t), \pi_j(t+1)), \quad \forall i \neq j, \forall t \tag{7}$$

e. 避免跟随冲突

如果机器人 i 在时间步 t 移动到顶点 v，则在下一个时间步 $t+1$ 机器人 j 不能移动到同一顶点 v，避免机器人彼此跟随:

$$\pi_j(t+1) \neq \pi_i(t), \quad \forall i \neq j, \forall t \tag{8}$$

f. 避免循环冲突

当每个机器人在同一时间步中移动到先前另一个机器人占据的顶点时，会造成循环冲突。为避免循环冲突，约束如下:

$$\pi_i(t+1) \neq \pi_j(t), \quad \pi_j(t+1) \neq \pi_i(t), \quad \forall i \neq j, \forall t \tag{9}$$

6. 优化模型的整合呈现

$$\text{minimize} \quad \sum_{1<i<k} |\pi_i| \tag{10}$$

$$\text{s.t.} \quad \begin{cases} \pi_i(t) \in F, & \forall i, \forall t \\ \pi_i(t+1) \in \{\pi_i(t) \cup a(\pi_i(t))\}, & \forall i, \forall t \\ \pi_i(t) \neq \pi_j(t), & \forall i \neq j, \forall t \\ (\pi_i(t), \pi_i(t+1)) \neq (\pi_j(t), \pi_j(t+1)), & \forall i \neq j, \forall t \\ \pi_j(t+1) \neq \pi_i(t), & \forall i \neq j, \forall t \\ \pi_i(t+1) \neq \pi_j(t), \quad \pi_j(t+1) \neq \pi_i(t), & \forall i \neq j, \forall t \end{cases} \tag{11}$$

5.1.2 A* 与 CBS

在上述基于多机器人路径规划的整数线性规划模型的基础上，为了进一步优化各机器人路径的规划和执行效率，我们引入了 A* 算法与 CBS 算法作为求解方式。A 算法是一种经典的启发式搜索算法，能够在单个机器人的路径规划中找到从起点到目标点的最优路径。然而，在多机器人系统中，由于存在路径冲突的问题，仅依靠 A 算法可能难以有效解决所有冲突。为此，我们采用了 CBS 算法，它能够在考虑多个机器人之间的路径冲突的同时，逐步优化和调整每个机器人的路径，使得最终的多机器人路径规划方案更加高效和合理，具体如下：

A* 算法是一种启发式搜索算法，通过选择启发式函数来指导搜索过程。A* 算法的核心是通过以下路径优劣评价公式来决定搜索路径：

$$f(x) = g(x) + h(x) \tag{12}$$

其中：

- $f(x)$ 是从初始状态经过状态 x 到目标状态的代价估计。
- $g(x)$ 是从初始状态到状态 x 的实际代价。
- $h(x)$ 是从状态 x 到目标状态的启发式估计代价。

启发式函数 $h(x)$ 通常选择曼哈顿距离，其计算公式为：

$$h(x) = |x_1 - t_1| + |x_2 - t_2| \tag{13}$$

其中 (x_1, x_2) 表示当前节点 x 的坐标，(t_1, t_2) 表示目标节点的坐标。

本题对应机器人运输时间最小化的计算步骤如下：

Step1. 初始化

将每个机器人的起点 s_i 设为初始节点，初始化 $g(s_i) = 0$，计算 $h(s_i)$，并将 $f(s_i) = g(s_i) + h(s_i)$ 放入优先队列（Open Set）。终点 t_i 设为每个机器人的目标节点。起点加入队列，Closed Set 用于记录已经经过的节点，防止重复搜索。

Step2. 选择节点

从 Open Set 中选择具有最小 f 值的节点 n，如果 $n = t$，即达到了目标点，算法结束，返回路径。

Step3. 扩展邻居节点

将当前节点 n 从 Open Set 中移除，加入 Closed Set，并遍历 n 的每个邻居节点 m。

Step4. 代价计算

a. 计算从当前节点 n 到邻居节点 m 的实际代价 $g(m)$：

$$g(m) = g(n) + \text{cost}(n, m) \tag{14}$$

b. 使用曼哈顿距离计算邻居节点 m 的启发式代价 $h(m)$。

c. 计算邻居节点 m 的总代价 $f(m) = g(m) + h(m)$。

Step5. 检查并更新节点

a. 如果邻居节点 m 已经在 Closed Set 中，跳过该节点。

b. 如果邻居节点 m 不在 Open Set 中，将其加入 Open Set，并记录从 n 到 m 的路径信息。

c. 如果邻居节点 m 已经在 Open Set 中，但新路径的实际代价 $g(m)$ 更小，则更新 $g(m)$ 和路径信息。

Step6. 处理冲突

检查邻居节点 m 是否与其他机器人发生冲突，如果发生冲突，则重新规划或调整路径。

Step7. 继续搜索

重复步骤 2 到步骤 6，直到找到目标节点 t 或者 Open Set 为空。如果找到目标点，则返回最优路径。

CBS 路径规划是一种用于多机器人路径规划的高效算法，尤其适用于解决多机器人之间路径冲突的问题。CBS 算法的核心思想是通过构建约束树（Constraint Tree, CT），分层次地解决路径规划中的冲突问题。算法可分为两层：

1. 底层：单机路径规划

对于每个机器人，使用 A* 算法进行单独的最短路径规划，以找到一个不考虑其他机器人的最优路径。

2. 顶层：冲突检测与处理

顶层会遍历底层规划出的路径，检查是否存在机器人之间的路径冲突。如果检测到冲突，则生成相应的约束，将冲突机器人重新分配到不同的路径，继续在底层进行规划，直到所有机器人路径均无冲突为止。

CBS 算法的执行步骤如下：

Step 1. 初始化

初始状态下，假设所有机器人之间没有冲突，为每个机器人生成一条最优路径。

Step 2. 冲突检测

检查所有机器人的路径，找出第一个发生冲突的位置。如果没有冲突，算法结束，返回所有机器人的路径。

Step 3. 生成约束

对于检测到的冲突，生成约束条件，以避免发生冲突的机器人在冲突时间访问冲突位置。

Step 4. 重新规划

在底层（A* 算法）中，使用新的约束条件重新规划冲突机器人的路径。

Step 5. 递归处理

返回顶层，重复冲突检测与处理过程，直到所有机器人路径无冲突为止。

Step 6. 输出结果

当所有机器人路径无冲突时，算法结束，返回所有机器人的无冲突路径。

通过 CBS 算法的分层处理机制，能够有效地解决多机器人路径规划中的冲突问题，保障各机器人能够安全、有效地到达目标位置。

5.1.3 模型求解

对于问题一中建立的基于多机器人路径规划的整数线性规划模型，我们采用了 A* 算法与 CBS（冲突分解搜索）算法相结合的方式进行求解。求解得三种不同地图下机器人运输的时间如下：

表 1: 三种地图下的机器人运输时间

地图类型	8*8 地图	16*16 地图	64*64 地图
总运输时间（单位：时间步）	46	136	2227

以 8*8 地图为例，A* 算法求解得到机器人运输时间是 46，其中八个机器人的运输路线方案具体如下：

表 2: 各机器人完整路径及时间步

机器人编号	完整路径	时间步
1	(1,7), (2,7), (2,6), (2,5), (2,4), (2,3), (3,3)	6
2	(5,0), (5,1), (5,2), (6,2), (6,3), (7,3), (7,4), (7,5)	7
3	(7,4), (7,3), (6,3), (5,3), (5,2)	4
4	(4,5), (4,6), (4,7), (5,7), (6,7)	4
5	(6,5), (5,5), (4,5), (3,5), (2,5), (1,5), (0,5), (0,6)	7
6	(5,2), (4,2), (3,2), (3,3), (3,4), (2,4)	5
7	(0,4), (0,3), (1,3), (2,3), (3,3), (4,3), (5,3), (6,3), (6,2), (7,2)	9
8	(4,1), (3,1), (2,1), (2,2), (1,2)	4

对于问题一的 A* 算法与 CBS 算法的算法复杂度分析如下:

A* 算法时间复杂度：

A* 算法在最坏情况下需要遍历所有结点，其时间复杂度与 Dijkstra 算法相同，为：

$$\mathcal{O}((V+E)\log V)$$

在通常情况下，A* 算法的时间复杂度为：

$$\mathcal{O}(b^d)$$

其中，b 为分支因子，d 为搜索深度。

分支因子 b 通常与可行的移动方向数量有关。若机器人可以在前后左右 5 个方向移动，假设有 n 个机器人，在每个节点，算法需要探索 5^n 个子节点。

搜索深度 d 为从起点到终点的最短路径深度。

因此，通常情况下 A* 算法的时间复杂度为：

$$\mathcal{O}(5^n \cdot d \cdot (n^2+1))$$

简化为：

$$\mathcal{O}(5^{n \cdot d} \cdot n^2)$$

A* 算法空间复杂度：

A* 算法使用两个列表来存储已经生成的和待计算的节点。在一般情况下，需要存储 V^n 个状态，因此算法的空间复杂度为：

$$\mathcal{O}(V^n)$$

其中，V 为节点总数。

CBS 算法时间复杂度：

CBS 算法在每次冲突解决时可能需要重新执行 A* 算法来计算机器人的路径。最坏情况下，每个机器人的路径都存在冲突。在理想情况下，每个机器人的路径都不冲突。因此，一般情况下，CBS 算法的时间复杂度为：

$$\mathcal{O}(n \cdot \mathcal{O}(A^*))$$

CBS 算法空间复杂度：

CBS 算法需要存储整个约束树，包括各种路径规划和约束。每个树节点都需要存储特定的路径信息和约束条件。因此，CBS 算法的空间复杂度为：

$$\mathcal{O}(n \cdot V)$$

5.2 问题二模型的建立与求解

5.2.1 多机器人终点驻留路径规划模型

问题 2 是在问题 1 的基础上假设每个机器人在完成任务后停留在目的地，在不发生冲突的前提下重新建立基于多机器人路径规划的整数线性规划模型，具体如下：

1. 目标函数

为了最小化每个智能体达到目标所需的时间步数之和，建立如下目标函数：

$$\text{minimize} \quad \sum_{1<i<k} |\pi_i| \tag{15}$$

其中，i 表示机器人，π_i 达到目标的时间步数。

2. 约束条件

a. 路径约束

每个机器人 i 的路径必须有效:

$$\pi_i(t) \in F, \quad \forall i, \forall t \tag{16}$$

其中，$\pi_i(t)$ 表示机器人 i 在时间步 t 时的位置，F 为可通行的顶点集合。

b. 移动约束

每个机器人每个时间步仅能移动到相邻的结点 v'，即仅能向前、后、左、右四个方向移动:

$$\pi_i(t+1) \in \{\pi_i(t) \cup a(\pi_i(t))\}, \quad \forall i, \forall t \tag{17}$$

其中，$a(\pi_i(t))$ 表示机器人 i 从顶点 $\pi_i(t)$ 移动到相邻顶点 v'。

c. 避免顶点冲突

在任何时间步 t 中，各机器人不能同时位于同一个顶点:

$$\pi_i(t) \neq \pi_j(t), \quad \forall i \neq j, \forall t \tag{18}$$

d. 避免边冲突

在任何时间步之间的移动中，各机器人不能同时经过同一条边:

$$(\pi_i(t), \pi_i(t+1)) \neq (\pi_j(t), \pi_j(t+1)), \quad \forall i \neq j, \forall t \tag{19}$$

e. 避免跟随冲突

如果机器人 i 在时间步 t 移动到顶点 v，则在下一个时间步 $t+1$ 机器人 j 不能移动到同一顶点 v，避免机器人彼此跟随:

$$\pi_j(t+1) \neq \pi_i(t), \quad \forall i \neq j, \forall t \tag{20}$$

f. 避免循环冲突

当每个机器人在同一时间步中移动到先前另一个机器人占据的顶点时，会造成循环冲突。为避免循环冲突，约束如下:

$$\pi_i(t+1) \neq \pi_j(t), \quad \pi_j(t+1) \neq \pi_i(t), \quad \forall i \neq j, \forall t \tag{21}$$

g. 机器人完成任务后停留在目的地

因假设每个机器人在完成任务后停留在目的地，因此可将任务完成后的机器人视为障碍物，在时间步 $|\pi_i|$ 后机器人的目标点 $t(i)$ 加入不可通行的顶点集合 O：

$$t(i) \in O, \quad \forall\, t > |\pi_i| \tag{22}$$

3. 优化模型的整合呈现

$$\text{minimize} \quad \sum_{1<i<k} |\pi_i| \tag{23}$$

$$\text{s.t.} \quad \begin{cases} \pi_i(t) \in F, & \forall i, \forall t \\ \pi_i(t+1) \in \{\pi_i(t) \cup a(\pi_i(t))\}, & \forall i, \forall t \\ \pi_i(t) \neq \pi_j(t), & \forall i \neq j, \forall t \\ (\pi_i(t), \pi_i(t+1)) \neq (\pi_j(t), \pi_j(t+1)), & \forall i \neq j, \forall t \\ \pi_j(t+1) \neq \pi_i(t), & \forall i \neq j, \forall t \\ \pi_i(t+1) \neq \pi_j(t), \quad \pi_j(t+1) \neq \pi_i(t), & \forall i \neq j, \forall t \\ t(i) \in O, & \forall t > |\pi_i| \end{cases} \tag{24}$$

5.2.2 模型求解

对于问题二中建立的多机器人终点驻留路径规划的整数线性规划模型，我们继续采用 A* 算法与 CBS 算法相结合的方式进行求解，求解得三种不同地图下机器人运输的时间如下：

表 3: 三种地图下的机器人运输时间

地图类型	8*8 地图	16*16 地图	64*64 地图
总运输时间（单位：时间步）	46	150	2229

以 8*8 地图为例，A* 算法求解得到机器人运输时间是 46，其中八个机器人的运输路线方案具体如下：

表 4: 各机器人完整路径及时间步

机器人编号	完整路径	时间步
1	(1,7), (2,7), (2,6), (2,5), (2,4), (2,3), (3,3)	6
2	(5,0), (5,1), (5,2), (6,2), (6,3), (7,3), (7,4), (7,5)	7
3	(7,4), (7,3), (6,3), (5,3), (5,2)	4
4	(4,5), (4,6), (4,7), (5,7), (6,7)	4
5	(6,5), (5,5), (4,5), (3,5), (2,5), (1,5), (0,5), (0,6)	7
6	(5,2), (4,2), (3,2), (3,3), (3,4), (2,4)	5
7	(0,4), (0,3), (1,3), (2,3), (3,3), (4,3), (5,3), (6,3), (6,2), (7,2)	9
8	(4,1), (3,1), (2,1), (2,2), (1,2)	4

此外，A* 算法与 CBS 算法的算法复杂度与问题一中一致。

5.3 问题三模型的建立与求解

5.3.1 基于多机器人路径规划的混合整数线性规划模型

问题三要求在问题二的基础上，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，确保完成机器人的任务调度且确保任务总运输时间最小，于是我们建立了一个混合整数线性规划模型 [2]。具体如下：

1. 模型建立与符号说明

a. $R = 1, 2, \ldots, n$：表示机器人集合，总数为 n。

b. $M = 1, 2, 3, \ldots, m$：表示任务集合，总数为 m。

c. $s(k)$ 和 $t(k)$ 分别表示任务 M_k 的出发点和目标点。

d. 二元变量 x_{ik}：表示机器人 R_i 是否执行任务 M_k。若 $x_{ik} = 1$，则任务 M_k 由机器人 R_i 执行。

e. π_i：表示机器人 R_i 的所有动作序列。

f. π_{ik}：表示机器人 R_i 执行任务 M_k 时的路径，即动作序列（包括机器人接受该任务时从当前地点到达任务 M_k 出发点的动作序列）。

g. $|\pi_{ik}|$：表示机器人 R_i 执行任务 M_k 所用的时间。

2. 目标函数

目标函数为最小化机器人的最大完工时间，即每个机器人配送完本身所有任务所需的时间步数之和：

$$\min \quad \max_{1 \le i \le n} \sum_{k=1}^{m} x_{ik} |\pi_{ik}| \tag{25}$$

3. 约束条件

a. 路径约束

每个机器人 i 的路径必须有效:

$$\pi_i(t) \in F, \quad \forall i, \forall t \tag{26}$$

其中，$\pi_i(t)$ 表示机器人 i 在时间步 t 时的位置，F 为可通行的顶点集合。

b. 移动约束

每个机器人每个时间步仅能移动到相邻的结点 v'，即仅能向前、后、左、右四个方向移动:

$$\pi_i(t+1) \in \{\pi_i(t) \cup a(\pi_i(t))\}, \quad \forall i, \forall t \tag{27}$$

其中，$a(\pi_i(t))$ 表示机器人 i 从顶点 $\pi_i(t)$ 移动到相邻顶点 v'。

c. 避免顶点冲突

在任何时间步 t 中，各机器人不能同时位于同一个顶点:

$$\pi_i(t) \neq \pi_j(t), \quad \forall i \neq j, \forall t \tag{28}$$

d. 避免边冲突

在任何时间步之间的移动中，各机器人不能同时经过同一条边:

$$(\pi_i(t), \pi_i(t+1)) \neq (\pi_j(t), \pi_j(t+1)), \quad \forall i \neq j, \forall t \tag{29}$$

e. 避免跟随冲突

如果机器人 i 在时间步 t 移动到顶点 v，则在下一个时间步 $t+1$ 机器人 j 不能移动到同一顶点 v，避免机器人彼此跟随:

$$\pi_j(t+1) \neq \pi_i(t), \quad \forall i \neq j, \forall t \tag{30}$$

f. 避免循环冲突

当每个机器人在同一时间步中移动到先前另一个机器人占据的顶点时，会造成循环冲突。为避免循环冲突，约束如下:

$$\pi_i(t+1) \neq \pi_j(t), \quad \pi_j(t+1) \neq \pi_i(t), \quad \forall i \neq j, \forall t \tag{31}$$

g. 机器人完成任务后停留在目的地

因假设每个机器人在完成任务后停留在目的地，因此可将任务完成后的机器人视为障碍物，在时间步 $|\pi_i|$ 后机器人的目标点 $t(i)$ 加入不可通行的顶点集合 O：

$$t(i) \in O, \quad \forall\, t > |\pi_i| \tag{32}$$

h. 保证每个任务都被运输

$$\sum_{i=1}^{n} x_{ik} \geq 1, \quad \forall k \in M \tag{33}$$

i. 每个机器人在时间 t 时刻最多执行一项任务

机器人 R_i 所有动作序列的时间步数之和等于机器人 R_i 执行所有任务所用的时间步长之和：

$$|\pi_i| = \sum_{k=1}^{m} x_{ik} |\pi_{ik}| \tag{34}$$

4. 优化模型的整合呈现

$$\min \quad \max_{1 \leq i \leq n} \sum_{k=1}^{m} x_{ik} |\pi_{ik}| \tag{35}$$

$$\text{s.t.} \quad \begin{cases} \pi_i(t) \in F, & \forall i, \forall t \\ \pi_i(t+1) \in \{\pi_i(t) \cup a(\pi_i(t))\}, & \forall i, \forall t \\ \pi_i(t) \neq \pi_j(t), & \forall i \neq j, \forall t \\ (\pi_i(t), \pi_i(t+1)) \neq (\pi_j(t), \pi_j(t+1)), & \forall i \neq j, \forall t \\ \pi_j(t+1) \neq \pi_i(t), & \forall i \neq j, \forall t \\ \pi_i(t+1) \neq \pi_j(t), \quad \pi_j(t+1) \neq \pi_i(t), & \forall i \neq j, \forall t \\ t(i) \in O, & \forall t > |\pi_i| \\ \displaystyle\sum_{i=1}^{n} x_{ik} \geq 1, & \forall k \in M \\ |\pi_i| = \displaystyle\sum_{k=1}^{m} x_{ik} |\pi_{ik}| \end{cases} \tag{36}$$

5.3.2 基于遗传算法的混合路径规划策略

在问题三的求解中，我们设计了一种基于遗传算法 [3] 的综合路径规划策略，结合了遗传算法、A* 算法和冲突分解搜索算法 (CBS)，用于确保机器人任务调度的最优化，并在不发生冲突的前提下最小化总运输时间。具体的求解过程如下：

1. 任务调度顺序的确定

首先，我们利用遗传算法来确定任务的最优调度顺序。遗传算法是一种模拟自然进化过程的启发式搜索算法，能够在复杂问题中找到接近最优的解。其具体步骤如下：

a. **初始化种群**：将机器人和任务集合初始化为种群中的个体，每个个体代表一个可能的任务分配方案。

b. **适应度函数**：设定适应度函数以衡量个体的优劣。适应度函数的目标是最小化机器人的最大完工时间。对于每个个体，我们计算其适应度值：

$$\text{Fitness} = \max_{1 \le i \le n} \sum_{k=1}^{m} x_{ik} |\pi_{ik}| \tag{37}$$

其中，x_{ik} 为二元变量，表示机器人 R_i 是否执行任务 M_k，$|\pi_{ik}|$ 为机器人 R_i 执行任务 M_k 所需的时间。

c. **选择、交叉和变异**：在多代的演化过程中，通过选择操作保留适应度高的个体，并通过交叉和变异操作生成新的个体，从而逐步优化任务调度顺序。最终，遗传算法将收敛到一个较优的调度顺序。

2. 最优路径的计算

在任务调度顺序确定后，我们使用 A* 算法计算每个任务的最优路径。A* 算法是一种启发式搜索算法，能够有效地找到从起点到目标点的最短路径。其具体步骤如下：

a. **路径代价评估**：A* 算法采用以下评估函数 $f(x)$ 来决定搜索路径：

$$f(x) = g(x) + h(x) \tag{38}$$

其中，$g(x)$ 为从起点到当前节点 x 的实际代价，$h(x)$ 为从当前节点 x 到目标节点的启发式估计代价。一般选择曼哈顿距离作为启发式函数 $h(x)$，即

$$h(x) = |x_1 - t_1| + |x_2 - t_2| \tag{39}$$

其中 (x_1, x_2) 表示当前节点 x 的坐标，(t_1, t_2) 表示目标节点的坐标。

b. **路径搜索与更新**：从起点开始，逐步扩展当前节点的邻居节点，计算其 $f(x)$ 值，并优先选择 $f(x)$ 最小的节点进行搜索，直到到达目标点。通过这种方式，可以为每个任务找到最优路径。

3. 路径冲突的解决

在多机器人路径规划中，由于机器人数量与任务数量不等，可能出现路径冲突的情况。为了解决此类冲突，我们引入了 CBS 算法。其步骤如下：

a. **初始路径生成**：为每个机器人生成不考虑其他机器人路径的最优路径。

b. **冲突检测与约束生成**：遍历所有机器人的路径，检测路径冲突。如果检测到冲突，则生成相应的约束条件，以避免冲突的发生。设定路径冲突的条件如下：

(1) **顶点冲突**：在任何时间步 t，各机器人不能同时位于同一个顶点：

$$\pi_i(t) \neq \pi_j(t), \quad \forall i \neq j, \forall t \tag{40}$$

(2) **边冲突**：在任何时间步之间的移动中，各机器人不能同时经过同一条边：

$$(\pi_i(t), \pi_i(t+1)) \neq (\pi_j(t), \pi_j(t+1)), \quad \forall i \neq j, \forall t \tag{41}$$

c. **路径重新规划**：根据生成的约束条件，重新规划冲突机器人的路径，并反复执行冲突检测与路径调整，直到所有冲突被消除。

4. 综合求解流程

最终，通过遗传算法确定任务的最优调度顺序，使用 A* 算法为每个任务求解最优路径，再结合 CBS 算法处理路径冲突，从而得到机器人系统的全局最优调度方案。最终目标是最小化每个机器人执行所有任务所需的时间步数之和：

$$\text{minimize} \quad \max_{1 \leq i \leq n} \sum_{k=1}^{m} x_{ik} |\pi_{ik}| \tag{42}$$

满足路径有效性、移动约束、冲突消除等约束条件，通过这一多算法混合求解策略，我们得到了在不发生冲突的前提下，机器人任务调度的全局最优解。

5.3.3 模型求解

对于问题三中建立的基于多机器人路径规划的混合整数线性规划模型，我们采用基于遗传算法的混合路径规划策略的方式进行求解，求解得三种不同地图下机器人运输的时间如下：

表 5: 三种地图下的机器人运输时间

地图类型	8*8 地图	16*16 地图	64*64 地图
总运输时间（单位：时间步）	65	201	1051

以 8*8 地图为例，A* 算法求解得到机器人运输时间是 46，其中八个机器人的运输路线方案具体如下：

表 6: 机器人任务的顺序

任务描述	任务顺序
机器人任务顺序	[6 1 2 5 3 4 0 7]

机器人的运输坐标如下所示：

表 7: 机器人位置顺序

位置序号	坐标	位置序号	坐标	位置序号	坐标
1	(0, 0)	2	(0, 1)	3	(0, 2)
4	(0, 3)	5	(0, 4)	6	(0, 3)
7	(0, 2)	8	(1, 2)	9	(2, 2)
10	(3, 2)	11	(4, 2)	12	(5, 2)
13	(6, 2)	14	(7, 2)	15	(6, 2)
16	(5, 2)	17	(5, 1)	18	(5, 0)
19	(5, 1)	20	(5, 2)	21	(5, 3)
22	(5, 4)	23	(5, 5)	24	(6, 5)
25	(7, 5)	26	(7, 4)	27	(7, 3)
28	(6, 3)	29	(5, 3)	30	(5, 2)
31	(4, 2)	32	(3, 2)	33	(2, 2)
34	(2, 3)	35	(2, 4)	36	(2, 5)
37	(3, 5)	38	(4, 5)	39	(4, 6)
40	(4, 7)	41	(5, 7)	42	(6, 7)
43	(6, 6)	44	(6, 5)	45	(5, 5)
46	(4, 5)	47	(3, 5)	48	(2, 5)
49	(1, 5)	50	(0, 5)	51	(0, 6)
52	(0, 7)	53	(1, 7)	54	(2, 7)
55	(2, 6)	56	(2, 5)	57	(2, 4)
58	(2, 3)	59	(3, 3)	60	(3, 2)
61	(3, 1)	62	(4, 1)	63	(3, 1)
64	(2, 1)	65	(2, 2)	66	(1, 2)

在问题三中，遗传算法的算法复杂度可以表示为迭代次数乘以操作算法复杂度和适应性函数复杂度之和：

- **迭代次数**：设算法迭代次数为 T，则需要执行 T 次迭代。时间复杂度为：

$$\mathcal{O}(T)$$

- **染色体长度**：染色体越长，每个个体的适应度计算时间和交叉、变异操作的时间都会增加。因此，时间复杂度与染色体长度成正比，为：

$$\mathcal{O}(L)$$

- **选择操作**：采用轮盘赌选择法，在最坏情况下需要遍历整个种群，因此时间复杂度为：

$$\mathcal{O}(N)$$

  其中，N 为种群大小。

- **交叉操作**：交叉操作在两个个体之间的基因交换，交叉操作本身的时间复杂度为 $\mathcal{O}(L)$，对于整个种群，时间复杂度为：

$$\mathcal{O}(L \times N)$$

- **变异操作**：变异操作在染色体的每一个基因上进行，因此时间复杂度为：

$$\mathcal{O}(L \times N)$$

综上，操作算法的时间复杂度为：

$$\mathcal{O}(L \times N)$$

适应度函数的复杂度：对于每次迭代中的所有个体，都需计算适应度值。设适应度函数的时间复杂度为 $\mathcal{O}(F(n))$，其计算复杂度为初始化时间与循环时间之和。设 A* 的时间复杂度为 $\mathcal{O}(A)$，则有：

$$F(n) = \mathcal{O}(A) + \mathcal{O}(n \times A) = \mathcal{O}(n \times A)$$

因此，遗传算法的总体时间复杂度为：

$$\mathcal{O}(T \times N \times (F(n) + L)) = \mathcal{O}(T \times N \times (n \times A + L))$$

遗传算法的空间复杂度包括种群存储、适应度值存储以及操作过程中的中间个体存储：

- **种群存储**：存储种群中所有个体的染色体信息，空间复杂度为：

$$\mathcal{O}(L \times N)$$

- **适应度值存储**：适应度值存储的数量为个体数量，空间复杂度为：

$$\mathcal{O}(N)$$

- **中间个体存储**：用于存储新一代的个体信息，数量相当于存储种群中所有个体的染色体信息，空间复杂度为：

$$\mathcal{O}(L \times N)$$

因此，遗传算法的总体空间复杂度为：

$$\mathcal{O}(L \times N)$$

5.4 问题四模型的建立与求解

5.4.1 基于多机器人路径规划的混合整数线性规划模型

在问题三的基础上，假设部分任务需要由指定的机器人来完成。这里任务完成的顺序与未被指定的任务调度不做约束。在不发生冲突的前提下，我们重新构建了混合整数规划模型，使以确保任务总运输时间最小，具体如下：

1. 模型建立与符号说明

a. $R = \{1, 2, \ldots, n\}$：表示机器人集合，总数为 n。

b. $M = \{1, 2, 3, \ldots, m\}$：表示任务集合，总数为 m。

c. $s(k)$ 和 $t(k)$ 分别表示任务 M_k 的出发点和目标点。

d. 二元变量 x_{ik}：表示机器人 R_i 是否执行任务 M_k。若 $x_{ik} = 1$，则任务 M_k 由机器人 R_i 执行。

e. π_i：表示机器人 R_i 的所有动作序列。

f. π_{ik}：表示机器人 R_i 执行任务 M_k 时的路径，即动作序列（包括机器人接受该任务时从当前地点到达任务 M_k 出发点的动作序列）。

g. $|\pi_{ik}|$：表示机器人 R_i 执行任务 M_k 所用的时间。

h. $B_i \subseteq M$：表示机器人 R_i 被分配的任务集合，其中的元素为任务集合 M 的子集。

2. 目标函数

目标函数为最小化机器人的最大完工时间，即每个机器人配送完本身所有任务所需的时间步数之和：

$$\text{minimize} \quad \max_{1 \le i \le n} \sum_{k=1}^{m} x_{ik} |\pi_{ik}| \tag{43}$$

3. 约束条件

在问题三的基础上，我们添加了以下约束条件：

a. 任务分配约束

机器人 R_i 必须完成其被分配的所有任务：

$$x_{ik} = 1, \quad \forall k \in B_i \tag{44}$$

b. 保证每个任务都被运输

$$\sum_{i=1}^{n} x_{ik} \ge 1, \quad \forall k \in M \tag{45}$$

c. 每个机器人在时间 t 时刻最多执行一项任务

机器人 R_i 所有动作序列的时间步数之和等于其执行所有任务所用的时间步长之和：

$$|\pi_i| = \sum_{k=1}^{m} x_{ik} |\pi_{ik}| \tag{46}$$

4. 优化模型的整合呈现

$$\text{minimize} \quad \max_{1 \le i \le n} \sum_{k=1}^{m} x_{ik} |\pi_{ik}| \tag{47}$$

$$
\text{s.t.} \quad \begin{cases}
\pi_i(t) \in F, & \forall i, \forall t, \\
\pi_i(t+1) \in \{\pi_i(t) \cup a(\pi_i(t))\}, & \forall i, \forall t, \\
\pi_i(t) \neq \pi_j(t), & \forall i \neq j, \forall t, \\
(\pi_i(t), \pi_i(t+1)) \neq (\pi_j(t), \pi_j(t+1)), & \forall i \neq j, \forall t, \\
\pi_j(t+1) \neq \pi_i(t), & \forall i \neq j, \forall t, \\
\pi_i(t+1) \neq \pi_j(t), \quad \pi_j(t+1) \neq \pi_i(t), & \forall i \neq j, \forall t, \\
t(i) \in O, & \forall t > |\pi_i|, \\
\sum_{i=1}^{n} x_{ik} \geq 1, & \forall k \in M \\
|\pi_i| = \sum_{k=1}^{m} x_{ik} |\pi_{ik}|, & \\
x_{ik} = 1, & \forall k \in B_i, \\
\text{minimize} \quad \max\limits_{1 \leq i \leq n} \sum\limits_{k=1}^{m} x_{ik} |\pi_{ik}| &
\end{cases} \tag{48}
$$

5.4.2 模型求解

对于问题四中建立的基于多机器人路径规划的混合整数线性规划模型，我们采用基于遗传算法的混合路径规划策略的方式进行求解，求解得三种不同地图下机器人运输的时间如下：

表 8: 三种地图下的机器人运输时间

地图类型	8*8 地图	16*16 地图	64*64 地图
总运输时间（单位：时间步）	46	150	1168

以 8*8 地图为例，A* 算法求解得到机器人运输时间是 46，其中八个机器人的运输路线方案具体如下：

表 9: 各机器人完整路径及时间步

机器人编号	完整路径	时间步
1	(1,7), (2,7), (2,6), (2,5), (2,4), (2,3), (3,3)	6
2	(5,0), (5,1), (5,2), (6,2), (6,3), (7,3), (7,4), (7,5)	7
3	(7,4), (7,3), (6,3), (5,3), (5,2)	4
4	(4,5), (4,6), (4,7), (5,7), (6,7)	4
5	(6,5), (5,5), (4,5), (3,5), (2,5), (1,5), (0,5), (0,6)	7
6	(5,2), (4,2), (3,2), (3,3), (3,4), (2,4)	5
7	(0,4), (0,3), (1,3), (2,3), (3,3), (4,3), (5,3), (6,3), (6,2), (7,2)	9
8	(4,1), (3,1), (2,1), (2,2), (1,2)	4

此外，遗传算法的算法复杂度与问题三中一致。

六　模型的优缺点及推广

6.1　模型的优点

1. 使用 A* 算法和 CBS 算法的组合能够有效解决多机器人路径规划问题。这种组合方法能够在确保全局最优解的同时减少路径冲突，优化了机器人之间的协调性，从而提高了整体系统的运行效率 。

2. 模型能够处理多种复杂场景，如机器人数量与任务数量不对等的情况。通过引入混合整数线性规划模型，能够在不增加计算复杂性的情况下，灵活应对不同的任务分配策略，确保每个任务都能够被有效完成 。

3. 该模型在各种不同的地图规模下均表现出色，能够适应从小规模到大规模地图的路径规划需求。这种适应性使得模型在实际应用中具有广泛的可操作性和实用性 。

4. 模型通过多种约束条件（如顶点冲突、边冲突等）有效地避免了机器人之间的路径冲突，确保了机器人在执行任务时的安全性和效率 。

5. 模型不仅可以解决基本的路径规划问题，还可以通过添加不同的约束条件来解决更复杂的调度问题，具有很强的扩展性和通用性 。

6.2　模型的缺点

1. 随着机器人数量和任务复杂性的增加，模型的计算复杂性迅速上升，尤其是在使用混合整数线性规划和遗传算法时，求解过程可能会变得非常耗时。这在大规模应用场景中可能导致实时性较差 。

2. 虽然 A* 算法和 CBS 算法在路径规划中表现良好，但它们依赖于启发式函数的设计，可能导致在某些情况下未能找到全局最优解。此外，启发式算法在处理高度复杂的场景时，可能会遇到局部最优问题 。

3. 模型假设机器人所处的环境是静态的，缺乏对动态环境的适应能力。如果环境中有不可预见的变化（如障碍物的出现或路径的突变），模型可能无法及时调整，导致规划结果失效 。

4. 在遗传算法的使用中，交叉概率、变异概率等参数的选择对模型的性能有很大影响。如果这些参数设置不当，可能导致收敛速度慢，甚至无法找到满意的解决方案 。

5. 虽然模型对机器人之间的路径冲突做了约束，但没有充分考虑其他实际操作中的约束条件，如能源消耗、机器人硬件限制等。这可能导致模型在实际应用中的效果不如预期 。

6.3　模型的推广

该模型虽然主要针对多机器人路径规划和调度问题，但其基本思想和方法具有广泛的推广应用潜力。除了在智能交通系统中用于多车辆路径优化和车辆调度外，还可以应用于仓储物流管理中的自动化仓储系统，以提高货物调度和路径规划的效率。此外，通过与深度学习和强化学习等人工智能技术相结合，模型在动态环境中的适应性和决策优化能力可以进一步提升，适用于无人机编队、机器人集群控制和应急救援等更复杂的领域。总之，该模型具备良好的扩展性和通用性，通过适应性调整，可在多个实际场景中发挥重要作用。

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] 张明佳. 混合整数规划方法的工程应用研究 [D]. 华中科技大学,2005.

[3] 吉根林. 遗传算法研究综述 [J]. 计算机应用与软件, 2004, 21(2):5.

附录 1 问题一代码

```
%A-star
import time as timer
import heapq
from itertools import product
import numpy as np
import copy


def move(loc, dir):
    directions = [(0, -1), (1, 0), (0, 1), (-1, 0), (0, 0)]
    return loc[0] + directions[dir][0], loc[1] + directions[dir][1]


def get_sum_of_cost(paths):
    rst = 0
    for path in paths:
        rst += len(path) - 1
        if (len(path) > 1):
            assert path[-1] != path[-2]
    return rst


def compute_heuristics(my_map, goal):
    # Use Dijkstra to build a shortest-path tree rooted at the goal location
    open_list = []
    closed_list = dict()
    root = {'loc': goal, 'cost': 0}
    heapq.heappush(open_list, (root['cost'], goal, root))
    closed_list[goal] = root
    while len(open_list) > 0:
        (cost, loc, curr) = heapq.heappop(open_list)
        for dir in range(4):
            child_loc = move(loc, dir)
            child_cost = cost + 1
            if child_loc[0] < 0 or child_loc[0] >= len(my_map) \
                    or child_loc[1] < 0 or child_loc[1] >= len(my_map[0]):
                continue
            if my_map[child_loc[0]][child_loc[1]]:
                continue
            child = {'loc': child_loc, 'cost': child_cost}
            if child_loc in closed_list:
                existing_node = closed_list[child_loc]
                if existing_node['cost'] > child_cost:
                    closed_list[child_loc] = child
```

```
                heapq.heappush(open_list, (child_cost, child_loc, child))
            else:
                closed_list[child_loc] = child
                heapq.heappush(open_list, (child_cost, child_loc, child))

    # build the heuristics table
    h_values = dict()
    for loc, node in closed_list.items():
        h_values[loc] = node['cost']
    return h_values


def get_location(path, time):
    if time < 0:
        return path[0]
    elif time < len(path):
        return path[time]
    else:
        return path[-1] # wait at the goal location


def get_path(goal_node, meta_agent):
    path = []
    for i in range(len(meta_agent)):
        path.append([])
    curr = goal_node
    while curr is not None:
        for i in range(len(meta_agent)):
            path[i].append(curr['loc'][i])
        curr = curr['parent']
    for i in range(len(meta_agent)):
        path[i].reverse()
        assert path[i] is not None

        print(path[i])

        if len(path[i]) > 1:
            # remove trailing duplicates
            while path[i][-1] == path[i][-2]:
                path[i].pop()
                print(path[i])
                if len(path[i]) <= 1:
                    break
            # assert path[i][-1] != path[i][-2] # no repeats at the end!!

    assert path is not None
```

```
    return path


class A_Star(object):

    def __init__(self, my_map, starts, goals, heuristics, agents, contraints):
        """my_map  - list of lists specifying obstacle positions
        starts     - [(x1, y1), (x2, y2), ...] list of start locations for CBS
        goals      - [(x1, y1), (x2, y2), ...] list of goal locations for CBS
        agents     - the agent (CBS) or meta-agent of the agent (MA-CBS) involved in collision
        constraints - list of dict constraints generated by a CBS splitter; dict = {agent,loc,timestep,positive
            }
        """

        self.my_map = my_map

        self.num_generated = 0
        self.num_expanded = 0
        self.CPU_time = 0

        self.open_list = []
        self.closed_list = dict()

        self.constraints = contraints # to be used to create c_table

        self.agents = agents

        # check if meta_agent is only a simple agent (from basic CBS)
        if not isinstance(agents, list):
            self.agents = [agents]
            # print(meta_agent)

            # add meta_agent keys to constraints
            for c in self.constraints:
                c['meta_agent'] = {c['agent']}

        # FILTER BY INDEX FOR STARTS AND GOALS AND HEURISTICS
        self.starts = [starts[a] for a in self.agents]
        self.heuristics = [heuristics[a] for a in self.agents]
        self.goals = [goals[a] for a in self.agents]

        self.c_table = [] # constraint table
        self.max_constraints = np.zeros((len(self.agents),), dtype=int)

    def push_node(self, node):
        f_value = node['g_val'] + node['h_val']
```

```
136
137         heapq.heappush(self.open_list, (f_value, node['h_val'], node['loc'], self.num_generated, node))
138         self.num_generated += 1
139
140     def pop_node(self):
141         _, _, _, id, curr = heapq.heappop(self.open_list)
142
143         self.num_expanded += 1
144         return curr
145
146     # return a table that constains the list of constraints of all agents for each time step.
147     def build_constraint_table(self, agent):
148         # constraint_table = {}
149         constraint_table = dict()
150
151         if not self.constraints:
152             return constraint_table
153         for constraint in self.constraints:
154             timestep = constraint['timestep']
155
156             t_constraint = []
157             if timestep in constraint_table:
158                 t_constraint = constraint_table[timestep]
159
160             # positive constraint for agent
161             if constraint['positive'] and constraint['agent'] == agent:
162                 t_constraint.append(constraint)
163                 constraint_table[timestep] = t_constraint
164             # and negative (external) constraint for agent
165             elif not constraint['positive'] and constraint['agent'] == agent:
166                 t_constraint.append(constraint)
167                 constraint_table[timestep] = t_constraint
168             # enforce positive constraints from other agents (i.e. create neg constraint)
169             elif constraint['positive']:
170                 neg_constraint = copy.deepcopy(constraint)
171                 neg_constraint['agent'] = agent
172                 # if edge collision
173                 if len(constraint['loc']) == 2:
174                     # switch traversal direction
175                     prev_loc = constraint['loc'][1]
176                     curr_loc = constraint['loc'][0]
177                     neg_constraint['loc'] = [prev_loc, curr_loc]
178                 neg_constraint['positive'] = False
179                 t_constraint.append(neg_constraint)
180                 constraint_table[timestep] = t_constraint
181
```

```
        return constraint_table

    # returns if a move at timestep violates a "positive" or a "negative" constraint in c_table
    def constraint_violated(self, curr_loc, next_loc, timestep, c_table_agent, agent):

        # print("the move : {}, {}".format(curr_loc, next_loc))

        if timestep not in c_table_agent:
            return None

        for constraint in c_table_agent[timestep]:

            if agent == constraint['agent']:
                # vertex constraint
                if len(constraint['loc']) == 1:
                    # positive constraint
                    if constraint['positive'] and next_loc != constraint['loc'][0]:
                        # print("time {} positive constraint : {}".format(timestep, constraint))
                        return constraint
                    # negative constraint
                    elif not constraint['positive'] and next_loc == constraint['loc'][0]:
                        # print("time {} negative constraint : {}".format(timestep, constraint))
                        return constraint
                # edge constraint
                else:
                    if constraint['positive'] and constraint['loc'] != [curr_loc, next_loc]:
                        # print("time {} positive constraint : {}".format(timestep, constraint))
                        return constraint
                    if not constraint['positive'] and constraint['loc'] == [curr_loc, next_loc]:
                        # print("time {} negative constraint : {}".format(timestep, constraint))
                        return constraint

        return None

    # returns whether an agent at goal node at current timestep will violate a constraint in next timesteps
    def future_constraint_violated(self, curr_loc, timestep, max_timestep, c_table_agent, agent):

        for t in range(timestep + 1, max_timestep + 1):
            if t not in c_table_agent:
                continue

            for constraint in c_table_agent[t]:

                if agent == constraint['agent']:
                    # vertex constraint
                    if len(constraint['loc']) == 1:
```

```
                    # positive constraint
                    if constraint['positive'] and curr_loc != constraint['loc'][0]:
                        # print("future time {} positive constraint : {}".format(t, constraint))
                        return True
                    # negative constraint
                    elif not constraint['positive'] and curr_loc == constraint['loc'][0]:
                        # print("time {} negative constraint : {}".format(timestep, constraint))
                        # print("future time {} negative constraint : {}".format(t, constraint))
                        return True

        return False

    def generate_child_nodes(self, curr):


        children = []
        ma_dirs = product(list(range(5)),
                          repeat=len(self.agents)) # directions for move() for each agent: 0, 1, 2, 3, 4


        for dirs in ma_dirs:
            # print(dirs)
            invalid_move = False
            child_loc = []
            # move each agent for new timestep & check for (internal) conflicts with each other
            for i, a in enumerate(self.agents):
                aloc = move(curr['loc'][i], dirs[i])
                # vertex collision; check for duplicates in child_loc
                if aloc in child_loc:
                    invalid_move = True
                    # print("internal conflict")
                    break
                child_loc.append(move(curr['loc'][i], dirs[i]))


            if invalid_move:
                continue

            for i, a in enumerate(self.agents):
                # edge collision: check for matching locs in curr_loc and child_loc between two agents
                for j, a in enumerate(self.agents):
                    if i != j:
                        # print(ai, aj)
                        if child_loc[i] == curr['loc'][j] and child_loc[j] == curr['loc'][i]:
                            invalid_move = True

```

```
            if invalid_move:
                continue

            # check map constraints and external constraints
            for i, a in enumerate(self.agents):
                next_loc = child_loc[i]
                # agent out of map bounds
                if next_loc[0] < 0 or next_loc[0] >= len(self.my_map) or next_loc[1] < 0 or next_loc[1] >= len(
                        self.my_map[0]):
                    invalid_move = True
                # agechild_locnt collison with map obstacle
                elif self.my_map[next_loc[0]][next_loc[1]]:
                    invalid_move = True
                # agent is constrained by a negative external constraint
                elif self.constraint_violated(curr['loc'][i], next_loc, curr['timestep'] + 1, self.c_table[i],
                                              self.agents[i]):
                    invalid_move = True
                if invalid_move:
                    break

            if invalid_move:
                continue

            # find h_values for current moves
            h_value = 0
            for i in range(len(self.agents)):
                h_value += self.heuristics[i][child_loc[i]]

            h_test = sum([self.heuristics[i][child_loc[i]] for i in range(len(self.agents))])

            assert h_value == h_test

            # g_value = curr['g_val']+ curr['reached_goal'].count(False)
            num_moves = curr['reached_goal'].count(False)
            # print("(edge) cost (curr -> child) in a* tree == ", num_moves)

            g_value = curr['g_val'] + num_moves

            reached_goal = [False for i in range(len(self.agents))]

            for i, a in enumerate(self.agents):

                if not reached_goal[i] and child_loc[i] == self.goals[i]:


                    if curr['timestep'] + 1 <= self.max_constraints[i]:
```

```
                    if not self.future_constraint_violated(child_loc[i], curr['timestep'] + 1,
                                                           self.max_constraints[i], self.c_table[i],
                                                           self.agents[i]):
                        reached_goal[i] = True
                else:
                    reached_goal[i] = True

        child = {'loc': child_loc,
                 'g_val': g_value, # number of new locs (cost) added
                 'h_val': h_value,
                 'parent': curr,
                 'timestep': curr['timestep'] + 1,
                 'reached_goal': copy.deepcopy(reached_goal)
                 }

        children.append(child)

    return children

def compare_nodes(self, n1, n2):
    """Return true is n1 is better than n2."""

    # print(n1['g_val'] + n1['h_val'])
    # print(n2['g_val'] + n2['h_val'])

    assert isinstance(n1['g_val'] + n1['h_val'], int)
    assert isinstance(n2['g_val'] + n2['h_val'], int)

    return n1['g_val'] + n1['h_val'] < n2['g_val'] + n2['h_val']

def find_paths(self):

    self.start_time = timer.time()

    print("> build constraint table")

    for i, a in enumerate(self.agents):
        table_i = self.build_constraint_table(a)
        print(table_i)
        self.c_table.append(table_i)
        if table_i.keys():
            self.max_constraints[i] = max(table_i.keys())

    h_value = sum([self.heuristics[i][self.starts[i]] for i in range(len(self.agents))])

    # assert h_value == h_test
```

```
366
367         root = {'loc': [self.starts[j] for j in range(len(self.agents))],
368                 # 'F_val' : h_value, # only consider children with f_val == F_val
369                 'g_val': 0,
370                 'h_val': h_value,
371                 'parent': None,
372                 'timestep': 0,
373                 'reached_goal': [False for i in range(len(self.agents))]
374                 }
375
376         # check if any any agents are already at goal loc
377         for i, a in enumerate(self.agents):
378             if root['loc'][i] == self.goals[i]:
379
380                 if root['timestep'] <= self.max_constraints[i]:
381                     if not self.future_constraint_violated(root['loc'][i], root['timestep'], self.
                            max_constraints[i],
382                                                            self.c_table[i], self.agents[i]):
383                         root['reached_goal'][i] = True
384
385                         self.max_constraints[i] = 0
386
387         self.push_node(root)
388         self.closed_list[(tuple(root['loc']), root['timestep'])] = [root]
389
390         while len(self.open_list) > 0:
391
392             # if num_node_generated >= 30:
393             #     return
394
395             curr = self.pop_node()
396
397
398             solution_found = all(curr['reached_goal'][i] for i in range(len(self.agents)))
399             # print(curr['reached_goal'] )
400             if curr['loc'] == self.goals and not solution_found:
401                 continue
402             if solution_found:
403                 return get_path(curr, self.agents)
404
405             children = self.generate_child_nodes(curr)
406
407             for child in children:
408
409                 f_value = child['g_val'] + child['h_val']
410
```

```
            # if (tuple(child['loc']),child['timestep']) in self.closed_list:
            #     existing_node = self.closed_list[(tuple(child['loc']),child['timestep'])]
            #     if self.compare_nodes(child, existing_node):
            #         self.closed_list[(tuple(child['loc']),child['timestep'])] = child
            #         self.push_node(child)
            # else:
            #     # print('bye child ',child['loc'])
            #     self.closed_list[(tuple(child['loc']),child['timestep'])] = child
            #     self.push_node(child)

            if (tuple(child['loc']), child['timestep']) in self.closed_list:
                existing = self.closed_list[(tuple(child['loc']), child['timestep'])]
                if (child['g_val'] + child['h_val'] < existing['g_val'] + existing['h_val']) and (
                        child['g_val'] < existing['g_val']) and child['reached_goal'].count(False) <=
                            existing[
                    'reached_goal'].count(False):
                    print("child is better than existing in closed list")
                    self.closed_list[(tuple(child['loc']), child['timestep'])] = child
                    self.push_node(child)
            else:
                # print('bye child ',child['loc'])
                self.closed_list[(tuple(child['loc']), child['timestep'])] = child
                self.push_node(child)

            # if (tuple(child['loc']),child['timestep']) not in self.closed_list:
            #     # existing_node = self.closed_list[(tuple(child['loc']),child['timestep'])]
            #     # if compare_nodes(child, existing_node):
            #     self.closed_list[(tuple(child['loc']),child['timestep'])] = child
            #     # print('bye child ',child['loc'])
            #     self.push_node(child)

        # if (tuple(curr['loc']),curr['timestep']) not in self.closed_list:
        #     self.closed_list[(tuple(curr['loc']),curr['timestep'])] = curr

    print('no solution')

    # print("\nEND OF A*\n") # comment out if needed
    return None
%CBS算法
import time as timer
import heapq
import random

import pandas as pd

# from single_agent_planner import compute_heuristics, a_star, get_location
```

```
# from multi_agent_planner import ll_solver, get_sum_of_cost, compute_heuristics, get_location

from a_star_class import A_Star, get_sum_of_cost, compute_heuristics, get_location
from openpyxl import load_workbook
import copy

import numpy

'''
   ## Reference to class
'''
######
'''
# Developer's cNOTE regarding Python's mutable default arguments:
#      The responsibiliy of preserving mutable values of passed arguments and
#      preventing retention of local mutable defaults by assigning immuatable default values (i.e. param=None)
     in parameters
#      is the responsiblity of the function being called upon
#      PEP 505 - None-aware operators: https://www.python.org/dev/peps/pep-0505/#syntax-and-semantics
'''

def generate_child(constraints, paths, agent_collisions, ma_list):

    assert isinstance(ma_list , list)

    collisions = detect_collisions(paths, ma_list)
    cost = get_sum_of_cost(paths)
    child_node = {
        'cost':cost,
        'constraints': copy.deepcopy(constraints),
        'paths': copy.deepcopy(paths), # {0: {'path':[..path...]}, ... , n: {'path':[..path...]} # not sure if
            other keys are needed
        'ma_collisions': collisions,
        'agent_collisions':copy.deepcopy(agent_collisions), # matrix of collisions in history between pairs of
            simple agents
        'ma_list': copy.deepcopy(ma_list) # [{a1,a2}, ... ]
    }
    return child_node

def detect_collision(path1, path2, pos=None):
    ##############################
    # Task 3.1: Return the first collision that occurs between two robot paths (or None if there is no
        collision)
    #         There are two types of collisions: vertex collision and edge collision.
    #         A vertex collision occurs if both robots occupy the same location at the same timestep
    #         An edge collision occurs if the robots swap their location at the same timestep.
```

```
    #         You should use "get_location(path, t)" to get the location of a robot at time t.
    assert pos is None
    if pos is None:
        pos = []
    t_range = max(len(path1),len(path2))
    for t in range(t_range):
        loc_c1 = get_location(path1,t)
        loc_c2 = get_location(path2,t)
        loc1 = get_location(path1,t+1)
        loc2 = get_location(path2,t+1)
        # vertex collision
        if loc1 == loc2:
            pos.append(loc1)
            return pos,t
        # edge collision
        if[loc_c1,loc1] ==[loc2,loc_c2]:
            pos.append(loc2)
            pos.append(loc_c2)
            return pos,t


    return None


def detect_collisions(paths, ma_list, collisions=None):
    ##############################
    # Task 3.1: Return a list of first collisions between all robot pairs.
    #         A collision can be represented as dictionary that contains the id of the two robots, the vertex
        or edge
    #         causing the collision, and the timestep at which the collision occurred.
    #         You should use your detect_collision function to find a collision between two robots.

    if collisions is None:
        collisions = []
    for ai in range(len(paths)-1):
        for aj in range(ai+1,len(paths)):
            if detect_collision(paths[ai],paths[aj]) !=None:
                position,t = detect_collision(paths[ai],paths[aj])

                # find meta-agents of agents in collision
                assert isinstance(ma_list , list)
                ma_i = get_ma_of_agent(ai, ma_list)
                assert isinstance(ma_list , list)
                ma_j = get_ma_of_agent(aj, ma_list)

                # check if internal collision in the same meta-agent
```

```
543                 if ma_i != ma_j:
544                     collisions.append({'a1':ai, 'ma1':ma_i,
545                                 'a2':aj, 'ma2':ma_j,
546                                 'loc':position,
547                                 'timestep':t+1})
548     return collisions
549 
550 def count_all_collisions_pair(path1, path2):
551     collisions = 0
552     t_range = max(len(path1),len(path2))
553     for t in range(t_range):
554         loc_c1 =get_location(path1,t)
555         loc_c2 = get_location(path2,t)
556         loc1 = get_location(path1,t+1)
557         loc2 = get_location(path2,t+1)
558         if loc1 == loc2 or [loc_c1,loc1] ==[loc2,loc_c2]:
559             collisions += 1
560     return collisions
561 
562 def count_all_collisions(paths):
563     collisions = 0
564     for i in range(len(paths)-1):
565         for j in range(i+1,len(paths)):
566             ij_collisions = count_all_collisions_pair(paths[i],paths[j])
567             collisions += ij_collisions
568 
569     # print("number of collisions: ", collisions)
570     return collisions
571 
572 def standard_splitting(collision, constraints=None):
573     ##############################
574     # Task 3.2: Return a list of (two) constraints to resolve the given collision
575     #         Vertex collision: the first constraint prevents the first agent to be at the specified location
        at the
576     #                         specified timestep, and the second constraint prevents the second agent to be at
        the
577     #                         specified location at the specified timestep.
578     #         Edge collision: the first constraint prevents the first agent to traverse the specified edge at
        the
579     #                       specified timestep, and the second constraint prevents the second agent to traverse
         the
580     #                       specified edge at the specified timestep
581     if constraints is None:
582         constraints = []
583 
584 
```

```
585     if len(collision['loc'])==1:
586         constraints.append({'agent':collision['a1'],
587                             'meta_agent': collision['ma1'],
588                             'loc':collision['loc'],
589                             'timestep':collision['timestep'],
590                             'positive':False
591                             })
592         constraints.append({'agent':collision['a2'],
593                             'meta_agent': collision['ma2'],
594                             'loc':collision['loc'],
595                             'timestep':collision['timestep'],
596                             'positive':False
597                             })
598     else:
599         constraints.append({'agent':collision['a1'],
600                             'meta_agent': collision['ma1'],
601                             'loc':[collision['loc'][0],collision['loc'][1]],
602                             'timestep':collision['timestep'],
603                             'positive':False
604                             })
605         constraints.append({'agent':collision['a2'],
606                             'meta_agent': collision['ma2'],
607                             'loc':[collision['loc'][1],collision['loc'][0]],
608                             'timestep':collision['timestep'],
609                             'positive':False
610                             })
611     return constraints
612 
613     # pass
614 
615 
616 def disjoint_splitting(collision, constraints=None):
617     ################################
618     # Task 4.1: Return a list of (two) constraints to resolve the given collision
619     #         Vertex collision: the first constraint enforces one agent to be at the specified location at the
620     #                           specified timestep, and the second constraint prevents the same agent to be at
        the
621     #                           same location at the timestep.
622     #         Edge collision: the first constraint enforces one agent to traverse the specified edge at the
623     #                         specified timestep, and the second constraint prevents the same agent to traverse
        the
624     #                         specified edge at the specified timestep
625     #         Choose the agent randomly
626     if constraints is None:
627         constraints = []
628 
```

```
    a = random.choice([('a1','ma1'), ('a2','ma2')]) # chosen agent
    agent = a[0]
    meta_agent = a[1]

    print(agent, collision)

    if len(collision['loc'])==1:
        constraints.append({'agent':collision[agent],
                            'meta_agent': collision[meta_agent],
                            'loc':collision['loc'],
                            'timestep':collision['timestep'],
                            'positive':True
                            })
        constraints.append({'agent':collision[agent],
                            'meta_agent': collision[meta_agent],
                            'loc':collision['loc'],
                            'timestep':collision['timestep'],
                            'positive':False
                            })
    else:
        if agent == 'a1':
            constraints.append({'agent':collision[agent],
                                'meta_agent': collision[meta_agent],
                                'loc':[collision['loc'][0],collision['loc'][1]],
                                'timestep':collision['timestep'],
                                'positive':True
                                })
            constraints.append({'agent':collision[agent],
                                'meta_agent': collision[meta_agent],
                                'loc':[collision['loc'][0],collision['loc'][1]],
                                'timestep':collision['timestep'],
                                'positive':False
                                })
        else:
            constraints.append({'agent':collision[agent],
                                'meta_agent': collision[meta_agent],
                                'loc':[collision['loc'][1],collision['loc'][0]],
                                'timestep':collision['timestep'],
                                'positive':True
                                })
            constraints.append({'agent':collision[agent],
                                'meta_agent': collision[meta_agent],
                                'loc':[collision['loc'][1],collision['loc'][0]],
                                'timestep':collision['timestep'],
                                'positive':False
                                })
```

```
    return constraints

# get the meta-agent an agent is a part of
# do NOT use for constraints, use key 'meta-agent' in constraint
def get_ma_of_agent(agent, ma_list):

    assert isinstance(ma_list , list)
    for ma in ma_list:
        # print(ma, ma_list)
        if agent in ma:
            # print(agent, ma)
            return ma
    raise BaseException('No meta-agent found for agent')


# find meta-agents of the agents that violates constraint
def meta_agents_violate_constraint(constraint, paths, ma_list, violating_ma=None):
    assert constraint['positive'] is True
    if violating_ma is None:
        violating_ma = []

    for i in range(len(paths)):
        ma_i = get_ma_of_agent(i, ma_list)

        if ma_i == constraint['meta_agent'] or ma_i in violating_ma:
            continue


        curr = get_location(paths[i], constraint['timestep'])
        prev = get_location(paths[i], constraint['timestep'] - 1)
        if len(constraint['loc']) == 1: # vertex constraint
            if constraint['loc'][0] == curr:
                # if ma_i not in violating_ma:
                    violating_ma.append(ma_i)
        else: # edge constraint
            if constraint['loc'][0] == prev or constraint['loc'][1] == curr \
                    or constraint['loc'] == [curr, prev]:
                # if ma_i not in violating_ma:
                violating_ma.append(ma_i)

    return violating_ma


def paths_violate_constraint(constraint, paths, rst=None):
    assert constraint['positive'] is True
    if rst is None:
```

```
        rst = []

    for i in range(len(paths)):
        if i == constraint['agent']:
            continue
        curr = get_location(paths[i], constraint['timestep'])
        prev = get_location(paths[i], constraint['timestep'] - 1)
        if len(constraint['loc']) == 1: # vertex constraint
            if constraint['loc'][0] == curr:
                rst.append(i)
        else: # edge constraint
            if constraint['loc'][0] == prev or constraint['loc'][1] == curr \
                    or constraint['loc'] == [curr, prev]:
                rst.append(i)
    return rst

def combined_constraints(constraints, new_constraints, updated_constraints=None):
    assert updated_constraints is None

    if isinstance(new_constraints, list):
        updated_constraints = copy.deepcopy(new_constraints)
    else:
        updated_constraints = [new_constraints]

    # print('combining constraints:')
    # print('const1: ', constraints)
    # print('const2: ', updated_constraints)

    for c in constraints:
        if c not in updated_constraints:
            updated_constraints.append(c)

    assert len(updated_constraints) <= len(constraints) + len(new_constraints)
    return updated_constraints


def bypass_found(curr_cost, new_cost, curr_collisions_num, new_collisions_num):
    if curr_cost == new_cost \
        and (new_collisions_num < curr_collisions_num):
        return True
    return False

def should_merge(collision, p, N=0):
    a1 = collision['a1']
    a2 = collision['a2']

```

```
    if a1 > a2:
        a1, a2 = a2, a1
    assert a1 < a2
    p['agent_collisions'][a1][a2] += 1

    if p['agent_collisions'][a1][a2] > N:
        return True

    ma1 = collision['ma1']
    ma2 = collision['ma2']

    # check it is same meta-agent
    assert ma1 != ma2
    assert not (a2 in ma1 or a1 in ma2)

    return False


class ICBS_Solver(object):
    """The high-level search of CBS."""

    def __init__(self, my_map, starts, goals):
        """my_map  - list of lists specifying obstacle positions
        starts    - [(x1, y1), (x2, y2), ...] list of start locations
        goals     - [(x1, y1), (x2, y2), ...] list of goal locations
        """

        self.my_map = my_map
        self.starts = starts
        self.goals = goals
        self.num_of_agents = len(goals)
        self.num_of_generated = 0
        self.num_of_expanded = 0
        self.CPU_time = 0

        self.open_list = []

        # compute heuristics for the low-level search
        self.heuristics = []
        for goal in self.goals:
            self.heuristics.append(compute_heuristics(my_map, goal))
        print(self.heuristics)

    def push_node(self, node):
        heapq.heappush(self.open_list, (node['cost'], len(node['ma_collisions']), self.num_of_generated, node))
        print("> Generate node {} with cost {}".format(self.num_of_generated, node['cost']))
```

```
        self.num_of_generated += 1


    def pop_node(self):
        _, _, id, node = heapq.heappop(self.open_list)
        print("> Expand node {} with cost {}".format(id, node['cost']))
        self.num_of_expanded += 1
        return node

    def empty_tree(self):
        self.open_list.clear()

    # algorithm for detecting cardinality
    # as 'non-cardinal' or 'semi-cardinal' or 'cardinal'
    # using standard splitting
    def detect_cardinal_conflict(self, AStar, p, collision):
        cardinality = 'non-cardinal'

        # temporary constraints (standard splitting) for detecting cardinal collision purposes
        temp_constraints = standard_splitting(collision)


        ma1 = collision['ma1'] #agent a1

        # print('Sending ma1 in collision {} to A* '.format(ma1))

        assert temp_constraints[0]['meta_agent'] == ma1
        path1_constraints = combined_constraints(p['constraints'], temp_constraints[0])
        astar_ma1 = AStar(self.my_map,self.starts, self.goals,self.heuristics,list(ma1),path1_constraints)
        alt_paths1 = astar_ma1.find_paths()

        # get current paths of meta-agent
        curr_paths = []
        for a1 in ma1:

            not_nested_list = p['paths'][a1]
            assert any(isinstance(i, list) for i in not_nested_list) == False


            curr_paths.append(p['paths'][a1])

        # print(curr_paths)
        # print(alt_paths1)

        # get costs for the meta agent
        curr_cost = get_sum_of_cost(curr_paths)
```

```
859
860         alt_cost = 0 # write inline if later
861         if alt_paths1:
862             alt_cost = get_sum_of_cost(alt_paths1)
863
864         # print('\t oldcost:{} newcost:{}'.format(curr_cost, alt_cost))
865
866         if not alt_paths1 or alt_cost > curr_cost:
867             cardinality = 'semi-cardinal'
868
869             print('alt_path1 takes longer or is empty. at least semi-cardinal.')
870
871
872         ma2 = collision['ma2'] #agent a2
873
874         # print('Sending ma2 in collision {} to A* '.format(ma2))
875
876         assert temp_constraints[1]['meta_agent'] == ma2
877         path2_constraints = combined_constraints(p['constraints'], temp_constraints[1])
878         astar_ma2 = AStar(self.my_map,self.starts, self.goals,self.heuristics,list(ma2),path2_constraints)
879         alt_paths2 = astar_ma2.find_paths()
880
881         # if not alt_path2 or bigger:
882         curr_paths = []
883         for a2 in ma2:
884             not_nested_list = p['paths'][a2]
885             assert any(isinstance(i, list) for i in not_nested_list) == False
886
887             curr_paths.append(p['paths'][a2])
888
889         # print(curr_paths)
890         # print(alt_paths2)
891
892         # get costs for the meta agent
893         curr_cost = get_sum_of_cost(curr_paths)
894
895         alt_cost = 0 # write inline if later
896         if alt_paths2:
897             alt_cost = get_sum_of_cost(alt_paths2)
898
899         # print('\t oldcost:{} newcost:{}'.format(curr_cost, alt_cost))
900
901         if not alt_paths2 or alt_cost > curr_cost:
902             # cardinality = 'semi-cardinal'
903             if cardinality == 'semi-cardinal':
904                 cardinality = 'cardinal'
```

```
            # print('identified cardinal conflict')

        else:
            cardinality = 'semi-cardinal'

            # print('alt_path2 takes longer or is empty. semi-cardinal.')
    # print('cardinality: ', cardinality)

    return cardinality

# returns new merged agents (the meta-agent), and updated list of ma_list
def merge_agents(self, collision, ma_list):

    # constraints = standard_splitting(collision)

    # collision simple agents and their meta-agent group
    a1 = collision['a1']
    a2 = collision['a2']
    ma1 = collision['ma1']
    ma2 = collision['ma2']

    meta_agent = set.union(ma1, ma2)

    print('new merged meta_agent ', meta_agent)

    assert meta_agent not in ma_list

    ma_list.remove(ma1)
    ma_list.remove(ma2)
    ma_list.append(meta_agent)

    return meta_agent, ma_list


def find_solution(self, disjoint):
    """ Finds paths for all agents from their start locations to their goal locations

    disjoint      - use disjoint splitting or not
    """

    self.start_time = timer.time()

    if disjoint:
        splitter = disjoint_splitting
    else:
```

```
951             splitter = standard_splitting
952 
953         AStar = A_Star
954 
955         # Generate the root node
956         # constraints - list of constraints
957         # paths       - list of paths, one for each agent
958         #               [[(x11, y11), (x12, y12), ...], [(x21, y21), (x22, y22), ...], ...]
959         # collisions   - list of collisions in paths
960         root = {
961             'cost':0,
962             'constraints': [],
963             'paths': [],
964             'ma_collisions': [],
965             'agent_collisions': None, # matrix of collisions in history between pairs of (meta-)agents
966             'ma_list': [] # [{a1,a2}, ... ]
967         }
968 
969         for i in range(self.num_of_agents): # Find initial path for each agent
970             astar = AStar(self.my_map, self.starts, self.goals, self.heuristics, [i], root['constraints'])
971             path = astar.find_paths()
972 
973 
974             if path is None:
975                 raise BaseException('No solutions')
976             root['ma_list'].append({i})
977             root['paths'].extend(path)
978 
979 
980 
981         root['cost'] = get_sum_of_cost(root['paths'])
982         root['ma_collisions'] = detect_collisions(root['paths'], root['ma_list'])
983         root['agent_collisions'] = numpy.zeros((self.num_of_agents, self.num_of_agents))
984         self.push_node(root)
985 
986 
987 
988         # ATTENTION: THE CBS LOOOOOOOOOOOP ==========@#Ⲩ#⁒#@Ⲩ@#⁒##@Ⲩ====== STARTS ---#Ⲩ⁒------ HERE ----
                @
989         # normal CBS with disjoint and standard splitting
990         while len(self.open_list) > 0:
991             if self.num_of_generated > 50000:
992                 print('reached maximum number of nodes. Returning...')
993                 return None
994             print('\n')
995             p = self.pop_node()
```

```
            if p['ma_collisions'] == []:
                self.print_results(p)
                # for pa in p['paths']:
                #     # print('asfasdfasdf   ',pa)
                return p['paths'], self.num_of_generated, self.num_of_expanded # number of nodes generated/
                    expanded for comparing implementations


            print('Node expanded. Collisions: ', p['ma_collisions'])
            for pa in p['paths']:
                print(pa)

            print('\n> Find Collision Type')

            # USING STANDARD SPLITTING
            # select a cardinal conflict;
            # if none, select a semi-cardinal conflict
            # if none, select a random conflict
            chosen_collision = None
            new_constraints = None
            collision_type = None
            for collision in p['ma_collisions']:

                print(collision)

                collision_type = self.detect_cardinal_conflict(AStar, p, collision)
                if collision_type == 'cardinal' and new_constraints is None:
                    print('Detected cardinal collision. Chose it.')
                    print(collision)

                    chosen_collision = collision
                    # collision_type = 'cardinal'
                    break

            else: # no cardinal collisions found
                for collision in p['ma_collisions']:
                    collision_type = self.detect_cardinal_conflict(AStar, p, collision)
                    if collision_type == 'semi-cardinal':

                        print('Detected semi-cardinal collision. Chose it.')
                        print(collision)
                        chosen_collision = collision
                        # collision_type = 'semi-cardinal'
                        break

                else: # no semi-cardinal collision found
```

```
            chosen_collision = p['ma_collisions'][0]
            assert chosen_collision is not None
            collision_type = 'non-cardinal'
            print('No cardinal or semi-cardinal conflict. Randomly choosing...')


        # keep track of collisions in history (aSh)
        chosen_a1 = chosen_collision['a1']
        chosen_a2 = chosen_collision['a2']
        if chosen_a1 > chosen_a2:
            # swap to only fill half of the matrix
            chosen_a1, chosen_a2 = chosen_a2, chosen_a1
        p['agent_collisions'][chosen_a1][chosen_a2] += 1


        new_constraints = splitter(chosen_collision)

        print('OLD CONSTS:')
        print(p['constraints'])

        print('NEW CONSTS:')
        print(new_constraints)
        print('\n')
        # child_nodes = None
        child_nodes = []
        assert child_nodes == []
        bypass_successful = False
        for constraint in new_constraints:
            print(constraint)

            updated_constraints = combined_constraints(p['constraints'], constraint)
            q = generate_child(updated_constraints, p['paths'], p['agent_collisions'], p['ma_list'])


            assert isinstance(p['ma_list'] , list)
            assert isinstance(q['ma_list'] , list)

            ma = constraint['meta_agent']

            print('\nSending meta_agent {} of constrained agent {} to A* '.format(ma, constraint['agent']))
            print('\twith constraints ', q['constraints'])

            for a in ma:
                print (q['paths'][a])

            astar = AStar(self.my_map,self.starts, self.goals,self.heuristics,list(ma),q['constraints'])
```

```
            paths = astar.find_paths()

            if paths is not None:

                for i in range(len(paths)):
                          print (paths[i])
                for i, agent in enumerate(ma):



                    not_nested_list = paths[i]
                    assert any(isinstance(j, list) for j in not_nested_list) == False


                    q['paths'][agent] = paths[i]

                if constraint['positive']:
                    # vol = paths_violate_constraint(constraint,q['paths'])
                    violating_ma_list = meta_agents_violate_constraint(constraint, q['paths'], q['ma_list'])
                    no_solution = False
                    for v_ma in violating_ma_list:

                        print('\nSending meta-agent violating constraint {} to A* '.format(v_ma))
                        print('\twith constraints ', q['constraints'])

                        for a in v_ma:
                            print (q['paths'][a])


                        v_ma_list = list(v_ma) # should use same list for all uses
                        astar_v_ma = AStar(self.my_map,self.starts,self.goals,self.heuristics,v_ma_list,q['
                            constraints'])
                        paths_v_ma = astar_v_ma.find_paths()



                        # replace paths of meta-agent with new paths found
                        if paths_v_ma is not None:

                            for i in range(len(v_ma_list)):
                                print (paths_v_ma[i])

                            for i, agent in enumerate(v_ma_list):

                                assert paths_v_ma[i] is not None
                                print(paths_v_ma[i])
```

```
                            not_nested_list = paths_v_ma[i]
                            assert any(isinstance(j, list) for j in not_nested_list) == False


                            q['paths'][agent] = paths_v_ma[i]
                        else:
                            print("no solution, moving on to next constraint")
                            no_solution = True
                            break # move on the next constraint

                    if no_solution:
                        continue # move on to the next constraint

                q['ma_collisions'] = detect_collisions(q['paths'],q['ma_list'])

                if chosen_collision in q['ma_collisions']:
                    print(q['paths'])
                    print('\nOH NO!!!!! chosen_collision is still in child :\'(')
                    print(chosen_collision)

                assert chosen_collision not in q['ma_collisions']

                q['cost'] = get_sum_of_cost(q['paths'])


                # assert that bypass is not possible if cardinal
                if collision_type == 'cardinal':
                    assert bypass_found(p['cost'], q['cost'], len(p['ma_collisions']), len(q['ma_collisions'
                        ])) == False

                # conflict should be resolved due to new constraints; compare costs and total number of
                     collisions
                if collision_type != 'cardinal' \
                        and bypass_found(p['cost'], q['cost'], len(p['ma_collisions']), len(q['ma_collisions'
                            ])):
                    print('> Take Bypass')
                    self.push_node(q)

                    bypass_successful = True
                    break # break out of constraint loop
                assert not bypass_successful
                child_nodes.append(copy.deepcopy(q))

        if bypass_successful:
            continue # start of while loop
```

```
            assert not bypass_successful

            # MA-CBS
            if should_merge(collision, p, 7):
                print('> Merge meta-agents into a new')
                # returns meta_agent, ma_list
                meta_agent, updated_ma_list = self.merge_agents(collision, p['ma_list'])


                # updated constraints
                updated_constraints = copy.deepcopy(p['constraints'])
                for c in updated_constraints:
                    if c['meta_agent'].issubset(meta_agent):
                        c['meta_agent'] = meta_agent

                print('Sending newly merged meta_agent {} to A* '.format(meta_agent))
                print('\twith constraints ', updated_constraints)

                for a in meta_agent:
                    print (p['paths'][a])


                # Update paths
                ma_astar = AStar(self.my_map,self.starts, self.goals,self.heuristics,list(meta_agent),
                    updated_constraints)
                ma_paths = ma_astar.find_paths()


                # if can be
                if ma_paths:

                    for i in range(len(meta_agent)):
                        print (ma_paths[i])

                    updated_paths = copy.deepcopy(p['paths'])

                    for i, agent in enumerate(meta_agent):

                        assert isinstance(i, int)

                        not_nested_list = ma_paths[i]
                        assert any(isinstance(j, list) for j in not_nested_list) == False
```

```
                    updated_paths[agent] = ma_paths[i]


                # for a in meta_agent:
                #     print (updated_paths[a])

                # Update collisions, cost
                updated_node = generate_child(updated_constraints, updated_paths, p['agent_collisions'],
                    updated_ma_list)


                # print('agents {}, {} merged into agent {}'.format(collision['a1'], a2, meta_agent))

                # Merge & restart
                # restart with only updated node with merged agents
                self.empty_tree()

                assert self.open_list == []

                self.push_node(updated_node)

                continue # start of while loop
            else:
                print("do not merge")

            assert len(child_nodes) <= 2
            print('bypass not found')
            for n in child_nodes:
                self.push_node(n)

        return None


    def print_results(self, node):
        print("\n Found a solution! \n")
        CPU_time = timer.time() - self.start_time
        print("CPU time (s): {:.2f}".format(CPU_time))
        print("Sum of costs: {}".format(get_sum_of_cost(node['paths'])))

        # file = "nodes-generated.csv"
        # result_file = open(file, "a", buffering=1)
        # result_file.write("{}\n".format(self.num_of_generated))

        print("Expanded nodes: {}".format(self.num_of_expanded))
        print("Generated nodes: {}".format(self.num_of_generated))
```

```
    print("Solution:")
    # 读取 Excel 文件中的指定工作表
    df = pd.read_excel('./results_第二问.xlsx', sheet_name='16x16map')
    df['位置列表'] = df['位置列表'].astype('object')

    print(node['paths'])
    for i in range(len(node['paths'])):
        print("agent", i, ": ", node['paths'][i])

    # 计算时间开销
    time = [len(node['paths'][i]) - 1 for i in range(len(node['paths']))]
    df['位置列表'] = node['paths']
    df['时间开销'] = time

    print(df)

    # Excel 文件名和工作表名
    filename = './results_第二问.xlsx'
    sheet_name = '16x16map'

    # 加载现有的 Excel 文件
    book = load_workbook(filename)

    # 使用 openpyxl 引擎并指定附加模式 ('a')
    with pd.ExcelWriter(filename, engine='openpyxl', mode='a', if_sheet_exists='replace') as writer:
        # 将 DataFrame 写入指定工作表，覆盖旧内容
        df.to_excel(writer, sheet_name=sheet_name, index=False)
```

附录 2 问题二代码

```
import argparse
import glob
from pathlib import Path
from cbs_basic import CBSSolver # original cbs with standard/disjoint splitting

# cbs with different improvements
from icbs_cardinal_bypass import ICBS_CB_Solver # only cardinal dectection and bypass
from icbs_complete import ICBS_Solver # all improvements including MA-CBS


from independent import IndependentSolver
from prioritized import PrioritizedPlanningSolver
from visualize import Animation
from single_agent_planner import get_sum_of_cost

HLSOLVER = "CBS"

LLSOLVER = "a_star"

def print_mapf_instance(my_map, starts, goals):
    print('Start locations')
    print_locations(my_map, starts)
    print('Goal locations')
    print_locations(my_map, goals)


def print_locations(my_map, locations):
    starts_map = [[-1 for _ in range(len(my_map[0]))] for _ in range(len(my_map))]
    for i in range(len(locations)):
        starts_map[locations[i][0]][locations[i][1]] = i
    to_print = ''
    for x in range(len(my_map)):
        for y in range(len(my_map[0])):
            if starts_map[x][y] >= 0:
                to_print += str(starts_map[x][y]) + ' '
            elif my_map[x][y]:
                to_print += '@ '
            else:
                to_print += '. '
        to_print += '\n'
    print(to_print)


def import_mapf_instance(filename):
```

```
    f = Path(filename)
    if not f.is_file():
        raise BaseException(filename + " does not exist.")
    f = open(filename, 'r')
    # first line: #rows #columns
    line = f.readline()
    rows, columns = [int(x) for x in line.split(' ')]
    rows = int(rows)
    columns = int(columns)
    # #rows lines with the map
    my_map = []
    for r in range(rows):
        line = f.readline()
        my_map.append([])
        for cell in line:
            if cell == '@':
                my_map[-1].append(True)
            elif cell == '.':
                my_map[-1].append(False)
    # #agents
    line = f.readline()
    num_agents = int(line)
    # #agents lines with the start/goal positions
    starts = []
    goals = []
    for a in range(num_agents):
        line = f.readline()
        sx, sy, gx, gy = [int(x) for x in line.split(' ')]
        starts.append((sx, sy))
        goals.append((gx, gy))
    f.close()
    return my_map, starts, goals


if __name__ == '__main__':
    result_file = open("results.csv", "w", buffering=1)
    nodes_gen_file = open("nodes-gen-cleaned.csv", "w", buffering=1)
    nodes_exp_file = open("nodes-exp-cleaned.csv", "w", buffering=1)

    instance = "./instances/16x16map.txt"
    input_instance = sorted(glob.glob(instance))

    print(input_instance)
    for file in input_instance:
        my_map, starts, goals = import_mapf_instance(file)
        print_mapf_instance(my_map, starts, goals)
```

```
91
92          print("***Run ICBS***")
93          cbs = ICBS_Solver(my_map, starts, goals)
94
95          solution = cbs.find_solution(True)
96
97          if solution is not None:
98              paths, nodes_gen, nodes_exp = [solution[i] for i in range(3)]
99              if paths is None:
100                 raise BaseException('No solutions')
101         else:
102             raise BaseException('No solutions')
103
104         cost = get_sum_of_cost(paths)
105         result_file.write("{},{}\n".format(file, cost))
106
107         nodes_gen_file.write("{},{}\n".format(file, nodes_gen))
108         nodes_exp_file.write("{},{}\n".format(file, nodes_exp))
109
110         print("***Test paths on a simulation***")
111         animation = Animation(my_map, starts, goals, paths)
112         animation.show()
113
114     result_file.close()
```

附录 3 问题三代码

```
from math import floor
import numpy as np
import time
import matplotlib.pyplot as plt # 导入所需要的库
from pathlib import Path

from test import heuristic, a_star_search
def import_mapf_instance(filename):
    f = Path(filename)
    if not f.is_file():
        raise BaseException(filename + " does not exist.")
    f = open(filename, 'r')
    # first line: #rows #columns
    line = f.readline()
    rows, columns = [int(x) for x in line.split(' ')]
    rows = int(rows)
    columns = int(columns)
    # #rows lines with the map
    my_map = []
    for r in range(rows):
        line = f.readline()
        my_map.append([])
        for cell in line:
            if cell == '@':
                my_map[-1].append(True)
            elif cell == '.':
                my_map[-1].append(False)
    # #agents
    line = f.readline()
    num_agents = int(line)
    # #agents lines with the start/goal positions
    starts = []
    goals = []
    for a in range(num_agents):
        line = f.readline()
        sx, sy, gx, gy = [int(x) for x in line.split(' ')]
        starts.append((sx, sy))
        goals.append((gx, gy))
    f.close()
    return my_map, starts, goals
class GA(object):
    def __init__(self, map, starts, goals,
                 maxgen=2000,
                 size_pop=100,
```

```
                cross_prob=0.80,
                pmuta_prob=0.02,
                select_prob=0.8):
        self.maxgen = maxgen # 最大迭代次数
        self.size_pop = size_pop # 群体个数
        self.cross_prob = cross_prob # 交叉概率
        self.pmuta_prob = pmuta_prob # 变异概率
        self.select_prob = select_prob # 选择概率

        self.goals = goals
        self.map = map
        self.starts = starts
        self.num = len(starts) # 城市个数 对应染色体长度
        self.heuristics = []

        # 通过选择概率确定子代的选择个数
        self.select_num = max(floor(self.size_pop * self.select_prob + 0.5), 2)

        # 父代和子代群体的初始化（不直接用np.zeros是为了保证单个染色体的编码为整数，np.zeros对应的数据类型为浮点
            型）
        self.chrom = np.array([0] * self.size_pop * self.num).reshape(self.size_pop,
                                                                      self.num) # 父 print(chrom.shape)(200, 14)
        self.sub_sel = np.array([0] * int(self.select_num) * self.num).reshape(self.select_num, self.num) # 子
            (160, 14)

        # 存储群体中每个染色体的路径总长度，对应单个染色体的适应度就是其倒数 #print(fitness.shape)#(200,)
        self.fitness = np.zeros(self.size_pop)

        self.best_fit = [] ##最优距离
        self.best_path = [] # 最优路径

    def rand_chrom(self):
        rand_ch = np.array(range(self.num))
        for i in range(self.size_pop):
            np.random.shuffle(rand_ch)
            self.chrom[i, :] = rand_ch
            self.fitness[i] = self.comp_fit(rand_ch)

    def comp_fit(self, one_path):
        time = 0
        path, res = a_star_search(self.map, (0, 0), self.starts[one_path[0]])
        time += res
        for i in range(self.num * 2 - 1):
            if i % 2 == 0:
                path, res = a_star_search(self.map, self.starts[one_path[int(i / 2)]], self.goals[one_path[int(i
                    / 2)]])
```

```
88          else:
89              path, res = a_star_search(self.map, self.goals[one_path[int(i / 2)]], self.starts[one_path[int(i
                    / 2 + 1)]])
90          time += res
91      return time
92
93  def select_sub(self):
94      fit = 1. / (self.fitness) # 适应度函数
95      cumsum_fit = np.cumsum(fit) # 累积求和 a = np.array([1,2,3]) b = np.cumsum(a) b=1 3 6
96      pick = cumsum_fit[-1] / self.select_num * (
97             np.random.rand() + np.array(range(int(self.select_num)))) # select_num 为子代选择个数 160
98      i, j = 0, 0
99      index = []
100     while i < self.size_pop and j < self.select_num:
101         if cumsum_fit[i] >= pick[j]:
102             index.append(i)
103             j += 1
104         else:
105             i += 1
106     self.sub_sel = self.chrom[index, :] # chrom 父
107
108 # 交叉，依概率对子代个体进行交叉操作
109 def cross_sub(self):
110     if self.select_num % 2 == 0: # select_num160
111         num = range(0, int(self.select_num), 2)
112     else:
113         num = range(0, int(self.select_num + 1), 2)
114     for i in num:
115         if self.cross_prob >= np.random.rand():
116             self.sub_sel[i, :], self.sub_sel[i + 1, :] = self.intercross(self.sub_sel[i, :], self.sub_sel[i
                    + 1, :])
117
118 def intercross(self, ind_a, ind_b): # ind_a, ind_b 父代染色体 shape=(1,14) 14=14个城市
119     r1 = np.random.randint(self.num) # 在num内随机生成一个整数 , num=14.即随机生成一个小于14的数
120     r2 = np.random.randint(self.num)
121     while r2 == r1: # 如果r1==r2
122         r2 = np.random.randint(self.num) # r2重新生成
123     left, right = min(r1, r2), max(r1, r2) # left 为r1,r2小值 , r2为大值
124     ind_a1 = ind_a.copy() # 父亲
125     ind_b1 = ind_b.copy() # 母亲
126     for i in range(left, right + 1):
127         ind_a2 = ind_a.copy()
128         ind_b2 = ind_b.copy()
129         ind_a[i] = ind_b1[i] # 交叉 （即ind_a （1,14） 中有个元素 和ind_b互换
130         ind_b[i] = ind_a1[i]
131         x = np.argwhere(ind_a == ind_a[i])
```

```
        y = np.argwhere(ind_b == ind_b[i])

        """
                下面的代码意思是 假如 两个父辈的染色体编码为【1234】，【4321】
                13421 23412
                交叉后为【1334】，【4221】
                交叉后的结果是不满足条件的，重复个数为2个
                需要修改为【1234】【4321】（即修改会来
        """
        if len(x) == 2:
            ind_a[x[x != i]] = ind_a2[i] # 查找ind_a 中元素=- ind_a[i] 的索引
        if len(y) == 2:
            ind_b[y[y != i]] = ind_b2[i]
    return ind_a, ind_b

# 变异模块 在变异概率的控制下，对单个染色体随机交换两个点的位置。
def mutation_sub(self):
    for i in range(int(self.select_num)): # 遍历每一个 选择的子代
        if np.random.rand() <= self.pmuta_prob: # 如果随机数小于变异概率
            r1 = np.random.randint(self.num) # 随机生成小于num==可设置 的数
            r2 = np.random.randint(self.num)
            while r2 == r1: # 如果相同
                r2 = np.random.randint(self.num) # r2再生成一次
            self.sub_sel[i, [r1, r2]] = self.sub_sel[i, [r2, r1]] # 随机交换两个点的位置。

# 进化逆转 将选择的染色体随机选择两个位置r1:r2 ， 将 r1:r2 的元素翻转为 r2:r1 ，如果翻转后的适应度更高，则替换
    原染色体，否则不变
def reverse_sub(self):
    for i in range(int(self.select_num)): # 遍历每一个 选择的子代
        r1 = np.random.randint(self.num) # 随机生成小于num==14 的数
        r2 = np.random.randint(self.num)
        while r2 == r1: # 如果相同
            r2 = np.random.randint(self.num) # r2再生成一次
        left, right = min(r1, r2), max(r1, r2) # left取r1 r2中小值，r2取大值
        sel = self.sub_sel[i, :].copy() # sel 为父辈染色体 shape=（1,14）

        sel[left:right + 1] = self.sub_sel[i, left:right + 1][::-1] # 将染色体中(r1:r2)片段 翻转为（r2:r1)
        if self.comp_fit(sel) < self.comp_fit(self.sub_sel[i, :]): # 如果翻转后的适应度小于原染色体，则不变
            self.sub_sel[i, :] = sel

# 子代插入父代，得到相同规模的新群体
def reins(self):
    index = np.argsort(self.fitness)[::-1] # 替换最差的（倒序）
    self.chrom[index[:self.select_num], :] = self.sub_sel

def info(self, one_path):
```

```
        Path = []
        time = 0
        path, res = a_star_search(self.map, (0, 0), self.starts[one_path[0]])
        Path.append(path)
        time += res
        for i in range(self.num * 2 - 1):
            if i % 2 == 0:
                path, res = a_star_search(self.map, self.starts[one_path[int(i / 2)]], self.goals[one_path[int(i
                    / 2)]])
            else:
                path, res = a_star_search(self.map, self.goals[one_path[int(i / 2)]],
                                          self.starts[one_path[int(i / 2 + 1)]])
            time += res
            Path.append(path)
        a = Path
        a = [a[i][1:] if i != 0 else a[i][:] for i in range(len(a))]
        b = []
        for z in a:
            for i in z:
                b.append(i)
        return b


if __name__ == '__main__':
    my_map, starts, goals = import_mapf_instance("./instances/16x16map.txt")
    print(my_map)
    module = GA(my_map, starts, goals)
    module.rand_chrom()
    for i in range(module.maxgen):
        module.select_sub() # 选择子代
        module.cross_sub() # 交叉
        module.mutation_sub() # 变异
        module.reverse_sub() # 进化逆转
        module.reins() # 子代插入

        for j in range(module.size_pop):
            module.fitness[j] = module.comp_fit(module.chrom[j])

        index = module.fitness.argmin()
        if (i + 1) % 10 == 0:
            print('第' + str(i + 1) + '代后的最短的路程: ' + str(module.fitness[index]))
            print('第' + str(i + 1) + '代后的最优路径:')

            print(module.chrom[index])
            path = module.info(module.chrom[index])
            print(path)
```

```
222
223         # 存储每一步的最优路径及距离
224         module.best_fit.append(module.fitness[index])
225         module.best_path.append(module.chrom[index])
226
227
228     print(module.best_path[module.size_pop - 1])
```

附录 4 问题四代码

```python
from math import floor
import numpy as np
import time
import matplotlib.pyplot as plt # 导入所需要的库
from pathlib import Path

from test import heuristic, a_star_search
def import_mapf_instance(filename):
    f = Path(filename)
    if not f.is_file():
        raise BaseException(filename + " does not exist.")
    f = open(filename, 'r')
    # first line: #rows #columns
    line = f.readline()
    rows, columns = [int(x) for x in line.split(' ')]
    rows = int(rows)
    columns = int(columns)
    # #rows lines with the map
    my_map = []
    for r in range(rows):
        line = f.readline()
        my_map.append([])
        for cell in line:
            if cell == '@':
                my_map[-1].append(True)
            elif cell == '.':
                my_map[-1].append(False)
    # #agents
    line = f.readline()
    num_agents = int(line)
    # #agents lines with the start/goal positions
    starts = []
    goals = []
    for a in range(num_agents):
        line = f.readline()
        sx, sy, gx, gy = [int(x) for x in line.split(' ')]
        starts.append((sx, sy))
        goals.append((gx, gy))
    f.close()
    return my_map, starts, goals


def distribute_tasks(num_tasks, num_robots):
    # Step 1: 生成任务列表并打乱顺序
```

```
    tasks = np.arange(num_tasks)
    np.random.shuffle(tasks)

    # Step 2: 随机分配每个机器人分配到的任务数量
    # 生成 num_robots 个随机数，这些数之和为 num_tasks
    split_points = np.sort(np.random.choice(range(1, num_tasks), num_robots - 1, replace=False))

    # Step 3: 根据分割点将任务分配给每个机器人
    task_distribution = np.split(tasks, split_points)

    return task_distribution

class GA(object):
    def __init__(self, map, starts, goals,
                 maxgen=2000,
                 size_pop=100,
                 cross_prob=0.80,
                 pmuta_prob=0.02,
                 select_prob=0.8):
        self.maxgen = maxgen # 最大迭代次数
        self.size_pop = size_pop # 群体个数
        self.cross_prob = cross_prob # 交叉概率
        self.pmuta_prob = pmuta_prob # 变异概率
        self.select_prob = select_prob # 选择概率

        self.goals = goals
        self.map = map
        self.starts = starts
        self.num = len(starts) # 城市个数 对应染色体长度

        # 通过选择概率确定子代的选择个数
        self.select_num = max(floor(self.size_pop * self.select_prob + 0.5), 2)

        # 父代和子代群体的初始化（不直接用np.zeros是为了保证单个染色体的编码为整数，np.zeros对应的数据类型为浮点型）
        # self.chrom = np.array([0] * self.size_pop * self.num).reshape(self.size_pop,
        #                                                               self.num) # 父 print(chrom.shape)(200, 14)
        # self.sub_sel = np.array([0] * int(self.select_num) * self.num).reshape(self.select_num, self.num) # 子 (160, 14)
        self.chrom = []
        self.sub_sel = []
        # 存储群体中每个染色体的路径总长度，对应单个染色体的适应度就是其倒数 #print(fitness.shape)#(200,)
        self.fitness = np.zeros(self.size_pop)

        self.best_fit = [] ##最优距离
        self.best_path = [] # 最优路径
```

```
    def rand_chrom(self):
        for i in range(self.size_pop):
            rand_ch = distribute_tasks(50, 5)

            self.fitness[i] = self.comp_fit(rand_ch)
            self.chrom.append(rand_ch)

    def comp_fit(self, one_path):
        total_time = 0
        start_pos = [(10, 20), (0, 0), (44, 28), (16, 41), (26, 45)]
        Path = []
        for i in range(5):
            path = []
            time = 0
            sub_path, res = a_star_search(self.map, start_pos[i], self.starts[one_path[i][0]])
            time += res
            path.append(sub_path)
            for j in range(len(one_path[i]) * 2 - 1):
                if j % 2 == 0:
                    sub_path, res = a_star_search(self.map, self.starts[one_path[i][int(j / 2)]], self.goals[
                        one_path[i][int(j / 2)]])
                else:
                    sub_path, res = a_star_search(self.map, self.goals[one_path[i][int(j / 2)]], self.starts[
                        one_path[i][int(j / 2 + 1)]])
                time += res
                path.append(sub_path)
            newPath = [path[k][1:] if k != 0 else path[k][:] for k in range(len(path))]
            path = []
            for sub_path in newPath:
                for pos in sub_path:
                    path.append(pos)
            total_time += time
            Path.append(path)


        return total_time

    def select_sub(self):
        fit = 1. / (self.fitness) # 适应度函数
        cumsum_fit = np.cumsum(fit) # 累积求和 a = np.array([1,2,3]) b = np.cumsum(a) b=1 3 6
        pick = cumsum_fit[-1] / self.select_num * (
               np.random.rand() + np.array(range(int(self.select_num)))) # select_num 为子代选择个数 160
        i, j = 0, 0
        index = []
        while i < self.size_pop and j < self.select_num:
            if cumsum_fit[i] >= pick[j]:
```

```
            index.append(i)
            j += 1
        else:
            i += 1
    self.sub_sel = [self.chrom[x] for x in index]

def cross_sub(self):
    if self.select_num % 2 == 0:
        num = range(0, int(self.select_num), 2)
    else:
        num = range(0, int(self.select_num + 1), 2)
    for i in num:
        if self.cross_prob >= np.random.rand():
            self.sub_sel[i], self.sub_sel[i + 1] = self.intercross(self.sub_sel[i], self.sub_sel[i + 1])

def intercross(self, ind_a, ind_b):
    offspring_a = []
    offspring_b = []

    for robot_tasks_a, robot_tasks_b in zip(ind_a, ind_b):
        num_tasks = len(robot_tasks_a)

        if num_tasks <= 1:
            offspring_a.append(robot_tasks_a.copy())
            offspring_b.append(robot_tasks_b.copy())
            continue

        # Select two distinct crossover points within the sublist
        r1, r2 = sorted(np.random.choice(range(num_tasks), 2, replace=False))

        # Create placeholders for offspring sublists
        child_a = [-1] * num_tasks
        child_b = [-1] * num_tasks

        # Copy the segment between crossover points from each parent
        child_a[r1:r2 + 1] = robot_tasks_a[r1:r2 + 1]
        child_b[r1:r2 + 1] = robot_tasks_b[r1:r2 + 1]

        # Fill in the remaining positions without duplicates
        remaining_tasks_a = [task for task in robot_tasks_b if task not in child_a]
        remaining_tasks_b = [task for task in robot_tasks_a if task not in child_b]

        # Fill child_a
        idx = (r2 + 1) % num_tasks
        for task in remaining_tasks_a:
            while child_a[idx] != -1:
```

```
                idx = (idx + 1) % num_tasks
            child_a[idx] = task

        # Fill child_b
        idx = (r2 + 1) % num_tasks
        for task in remaining_tasks_b:
            while child_b[idx] != -1:
                idx = (idx + 1) % num_tasks
            child_b[idx] = task

        offspring_a.append(child_a)
        offspring_b.append(child_b)

    return offspring_a, offspring_b

def mutation_sub(self):
    for i in range(int(self.select_num)):
        if np.random.rand() <= self.pmuta_prob:
            robot_idx = np.random.randint(0, 5)
            if len(self.sub_sel[i][robot_idx]) > 1:
                task_idx1, task_idx2 = np.random.choice(len(self.sub_sel[i][robot_idx]), 2, replace=False)
                self.sub_sel[i][robot_idx][task_idx1], self.sub_sel[i][robot_idx][task_idx2] = \
                    self.sub_sel[i][robot_idx][task_idx2], self.sub_sel[i][robot_idx][task_idx1]

def reverse_sub(self):
    for i in range(int(self.select_num)):
        robot_idx = np.random.randint(0, 5)
        tasks = self.sub_sel[i].copy() # Create a copy to avoid unintended side-effects
        if len(tasks) > 1 and isinstance(tasks[robot_idx], list):
            if len(tasks[robot_idx]) > 1:
                r1, r2 = sorted(np.random.choice(len(tasks[robot_idx]), 2, replace=False))
                # Reverse the selected segment
                tasks[robot_idx][r1:r2 + 1] = tasks[robot_idx][r1:r2 + 1][::-1]
                # Compare fitness and replace if improved
                if self.comp_fit(tasks) < self.comp_fit(self.sub_sel[i]):
                    self.sub_sel[i] = tasks

def reins(self):
    index = np.argsort(self.fitness)[::-1] # 替换最差的
    for i in range(self.select_num):
        self.chrom[index[i]] = self.sub_sel[i]


def info(self, one_path):
    Path = []
    time = 0
```

```
225         path, res = a_star_search(self.map, (0, 0), self.starts[one_path[0]])
226         Path.append(path)
227         time += res
228         for i in range(self.num * 2 - 1):
229             if i % 2 == 0:
230                 path, res = a_star_search(self.map, self.starts[one_path[int(i / 2)]], self.goals[one_path[int(i
                        / 2)]])
231             else:
232                 path, res = a_star_search(self.map, self.goals[one_path[int(i / 2)]],
233                                           self.starts[one_path[int(i / 2 + 1)]])
234             time += res
235             Path.append(path)
236         a = Path
237         a = [a[i][1:] if i != 0 else a[i][:] for i in range(len(a))]
238         b = []
239         for z in a:
240             for i in z:
241                 b.append(i)
242         return b
243
244
245 if __name__ == '__main__':
246     my_map, starts, goals = import_mapf_instance("./instances/64x64map.txt")
247     print(my_map)
248     module = GA(my_map, starts, goals)
249     module.rand_chrom()
250     for i in range(module.maxgen):
251         module.select_sub() # 选择子代
252         module.cross_sub() # 交叉
253         module.mutation_sub() # 变异
254         module.reverse_sub() # 进化逆转
255         module.reins() # 子代插入
256
257         for j in range(module.size_pop):
258             module.fitness[j] = module.comp_fit(module.chrom[j])
259
260         index = module.fitness.argmin()
261         if (i + 1) % 10 == 0:
262             print('第' + str(i + 1) + '代后的最短的路程: ' + str(module.fitness[index]))
263             print('第' + str(i + 1) + '代后的最优路径:')
264
265             print(module.chrom[index])
266             # path = module.info(module.chrom[index])
267             # print(path)
268
269         # 存储每一步的最优路径及距离
```

```
270         module.best_fit.append(module.fitness[index])
271         module.best_path.append(module.chrom[index])
272 
273 
274     print(module.best_path[module.size_pop - 1])
```