1. 基本思路

核心问题：让多个机器人在大楼里尽快完成任务

简单流程：

接任务 → 找机器人 → 规划路径 → 执行任务

2. 三个主要模块

2.1 地图模块

把大楼变成"图"，每个位置是点，通道是边

三种通道：

普通路：同层移动

楼梯：层间移动，速度慢但稳定

电梯：层间移动，速度快但不稳定

2.2 调度模块

任务发布

找能干的，离得最近的机器人

规划路线

2.3 规划模块

记录电梯被占用时间和状态

1. **楼梯（简单）**

楼梯的时间 = 楼梯长度 / 机器人爬楼速度

2. **电梯（复杂点）**

电梯的特殊性：

不是匀速：1 楼到 3 楼的时间 ≠ 1 到 2 楼时间的 2 倍

容量有限：一次只能装 1 个机器人

需要等待：不能随到随用

方案：

1. 计算电梯时间

提前测量好电梯时间

电梯时间表 = {

等待：

(1,2)/(2,1)：运行 1.75s,

(1,3)/(3,1)：运行 3.5s

…

使用：

(1,2)：开门 1.5s +关门 1.5s. + 运行 1.75s + 开门 1.5s = 6.25s, # 1 楼到 2 楼

(1,2),(2,3)：开门 1.5s +关门 1.5s + 运行 1.75s + 开门 1.5s + 关门 1.5s. + 运行 1.75s + 开门 1.5s = 11s # 1 楼到 2 楼， 2 楼再到 3 楼

(1,3)：开门 1.5s +关门 1.5s. + 运行 3.5s + 开门 1.5s = 8s, # 1 楼到 3 楼

…

}

2. 按时间表管理电梯

记录每个电梯(编号 1-6)的实际使用情况及计划被占用的时间及空闲时位置信息， 给每个电梯都设置一个 schedule

3. 冲突检测

Def 电梯是否可用(电梯编号, 起点层, 终点层, 想用的时间, 要用多久):

检查这个时间段内，有没有其他机器人用同一段电梯

for (已用开始, 已用结束) in 电梯占用记录:

if 时间重叠(想用的时间, 想用的时间+要用多久, 已用开始, 已用结束):

return False，waiting time # 冲突及等待时间

return True，waiting arriving time # 可以用, 电梯到达机器人楼层的时间

4. 实际使用流程

从机器人起点开始，计算使用楼梯到达任务地点的最短时间

对于每台电梯(1-6)：

从机器人起点开始，假设电梯正处于空闲状态且位于该楼层等待机器人→ 计算机器人到达电梯口需要的时间→查时间表计算要多久→电梯到达的时间→计算机器人到达任务地点的最短时间

→检查这段时间电梯是否空闲 → 空闲则得出使用电梯方法到达任务地点最短时间

→如果不空闲，则计算等待电梯空闲所需要的时间得出使用电梯方法到达任务地点最短时间

比较使用楼梯和电梯两种方法所用时间， 输出规划路径，前往任务地点。

Algorithm 1: Multi-Robot Task Scheduling and Path Planning in Multi-Floor Building

Input: Task set $T = \{t_1, t_2, ...\}$, Robot set $R = \{r_1, r_2, ...\}$, Elevator set $E = \{e_1, ..., e_6\}$

Output: Optimal path plan for each task

Module 1: Graph initial

1: Initialize building graph G(V, E)

2: For each floor f and connection c:

3: Add node v(f, c) and edge weight w according to travel distance and type

4: For each elevator $e_i \in E$:

5: Initialize e_i.state ← idle, e_i.current_floor ← 1, e_i.schedule ← ∅

Module 2: Task Assignment

6: For each task $t \in T$:

7: candidate_robots ← {r ∈ R | r.skill = t.skill}

8: For each r ∈ candidate_robots:

9: time_estimate ← estimate_travel_time(r.position, t.start)

10: total_cost ← r.available_time + time_estimate

11: Select r* = argmin(total_cost)

12: Assign task t → r*

Module 3: Path plan and State Update

13: For each assigned (r*, t):

14: best_time ← ∞

15: best_path ← ∅

--- Option A: Use Stairs ---

16: stair_time ← compute_stair_time(r*.position, t.target)

17: if stair_time < best_time:

18: best_time ← stair_time

19: best_path ← "stairs"

--- Option B: Use Elevator ---

20: For each elevator e ∈ E:

21: from_floor ← floor(r*.position)

22: to_floor ← floor(t.target)

23: t_move_to_e ← travel_time_to_elevator(r*, e)

24: travel_time ← lookup(ElevatorTable, from_floor, to_floor)

25: desired_start ← r*.available_time + t_move_to_e

26: (available, wait_time) ← check_elevator_conflict(e, desired_start, travel_time)

27: total_time ← t_move_to_e + wait_time + travel_time +
travel_time_exit(e, t.target)

28: if total_time < best_time:

29: best_time ← total_time

30: best_path ← "elevator e.id"

31: Update r*.available_time ← r*.available_time + best_time

32: Update r*.position ← t.target

33: If best_path involves elevator e:

34: Append (start_time, end_time, from_floor, to_floor, r*.id) to e.schedule

35: Output: Task t executed by r* using best_path with time best_time