智能加工系统中 RGV 的动态调度模型

摘要

本文基于**遗传算法**与**离散粒子群算法**分别分析了在一道工序与两道工序下的 RGV 动态调度策略，同时，我们还利用**计算机仿真**模拟了系统在 CNC 发生故障时的作业结果，并利用基于概率统计的**蒙特卡洛算法**评估了此时调度算法的性能。

针对第一种情况，由于一道工序的物料加工可以在任意一台 CNC 上完成，因此我们主要分析了多台 CNC 请求时 RGV 的两种调度策略。一种是**“先请求先服务”**，即 RGV 优先服务更早发出需求信号的 CNC；另一种则是**“就近原则”**，他指的是 RGV 总是做出当前情况下最好的选择：优先服务距离更近、上下料时间更短的 CNC。

在以上两种调度策略的基础上，我们利用遗传算法进行了优化验证。如果把每台 CNC 的每次加工过程都看做一个节点，那么最优的调度策略就是一条遍历所有节点一次且耗费总时间最少的路径，而这恰好就是一个经典的**旅行商（TSP）问题**。接下来我们利用遗传算法求解出了它的最优路径，而其所对应的最优调度策略恰好是 RGV **从** CNC1 **顺序作业到** CNC8 **的循环**，且三组数据对应的作业效率分别为 $\mathbf{93.1}\%$ 、$\mathbf{90.6}\%$、与 $\mathbf{92.7}\%$。

针对第二种情况，我们首先遍历了 8 台 CNC 的所有可能组合情况（即哪些负责第一道工序，哪些负责第二道），由此得到了最优的组合方式（此时的系统在一个轮次内加工完成的物料数最多）：第一组数据是 CNC$\mathbf{1,3,5,7}$ 执行第一道工序，CNC$\mathbf{2,4,6,8}$ 执行第二道工序；第二组数据是 CNC$\mathbf{2,4,6,8}$ 执行第一道工序，CNC$\mathbf{1,3,5,7}$ 执行第二道工序；第三组数据则是 CNC$\mathbf{1,2,4,6,7}$ 执行第一道工序，CNC$\mathbf{3,5,8}$ 执行第二道工序。考虑到 CNC 编号的离散性，我们改进了粒子群算法的运动规则并将其应用于最优调度策略的求解（具体调度策略参见 Excel 表格），并最终得到了三组数据对应的作业效率分别为 $\mathbf{85.4}\%$ 、$\mathbf{71.4}\%$、与 $\mathbf{66.6}\%$。

针对第三种情况，考虑到故障发生的随机性，我们不再利用遗传算法或离散粒子群算法求解全局的最优调度策略，而只是分析了 RGV 在 CNC 发生故障时的响应机制：它会移动到距离故障机器最近的其它机器处作业。同时，我们还基于蒙特卡洛算法对该机制进行了性能评估。

最后，在本题一辆 RGV 的智能加工系统基础上，我们还通过**时间窗模型**建立了一套多个 RGV 的加工系统的调度模型。

关键字： 遗传算法　离散粒子群算法　旅行商问题　计算机仿真　蒙特卡洛算法

一、 问题的重述

1.1 问题的背景

现如今，随着自动化产业的不断发展，智能 RGV 系统越来越广泛地应用于零件加工与装配领域。尽管 RGV 具有运行速度快、性能稳定性好等特点，但如何高效地对其进行动态调度，以缩短整个加工作业的时间，成为了亟待解决的问题。在本题中，我们考虑这样一个智能加工系统：它由 1 辆轨道式自动引导车（RGV）、1 条 RGV 直线轨道、8 台计算机数控机床（CNC）、1 条上料传送带以及 1 条下料传送带等附属设备构成。其中 RGV 自带机械臂与清洗槽，能够完成上下料与清洗物料的任务，且其装载的智能控制模块也能帮助它发送与接收指令。

1.2 问题的提出

现在我们考虑以下三种具体的加工作业情况：

- 一道工序的物料加工，即每台 CNC 安装同样的刀具，物料可以在任一台 CNC 上加工完成。
- 两道工序的物料加工，即每个物料的第一和第二道工序分别由两台不同的 CNC 依次加工完成。
- 考虑 CNC 在加工过程中可能发生故障（发生概率约为 1%）的情况，每次故障排除（未完成的物料报废）的时间介于 10 至 20 分钟之间，故障排除后即刻加入作业序列。要求同时考虑一道工序和两道工序的物料加工作业情况。

针对以上每种情况，我们需要完成以下任务：

1. 对于一般条件下的智能系统加工情况，分析 RGV 的动态调度模型和相应的求解算法。

2. 分别考虑以下三种具体的加工情况，利用所给的相应系统作业参数对模型的实用性和算法的有效性进行检验，并分析不同情况下 RGV 的调度策略和系统的作业效率，最后给出具体的结果。

二、 模型的假设

1. 忽略上料传送带与下料传送带的传动时间；
2. 忽略机械臂从上料传送带抓取生料到机械手爪里所需的时间；
3. 假设 RGV 会按照设定好的调度策略分毫不差地运行；
4. 取 CNC 的故障发生概率为其统计结果：1%；

5. 假设 CNC 的故障排除时间服从于 [600 秒,1200 秒] 上的均匀分布。

三、 符号说明

表 1　符号说明

符号	说明	单位
$\boldsymbol{CNCi}$	第 i 台 CNC	-
$\boldsymbol{t_{m1}}$	RGV 移动 1 个单位所需时间	秒
$\boldsymbol{t_{m2}}$	RGV 移动 2 个单位所需时间	秒
$\boldsymbol{t_{m3}}$	RGV 移动 3 个单位所需时间	秒
$\boldsymbol{t_p}$	CNC 加工完成一个一道工序的物料所需时间	秒
$\boldsymbol{t_r}$	RGV 为 CNC1，3，5，7 一次上下料所需时间	秒
$\boldsymbol{t_l}$	RGV 为 CNC2，4，6，8 一次上下料所需时间	秒
$\boldsymbol{t_{e1}}$	CNC 加工完成一个两道工序物料的第一道工序所需时间	秒
$\boldsymbol{t_{e2}}$	CNC 加工完成一个两道工序物料的第二道工序所需时间	秒
$\boldsymbol{t_c}$	RGV 完成一个物料的清洗作业所需时间	秒
$\boldsymbol{t_w}$	RGV 等待某一个 CNC 完成加工作业的时间	秒
$\boldsymbol{n}$	智能加工系统在一个班次内的物料加工数量	个
$\boldsymbol{t_{ij}}$	从 CNCi 开始加工物料到 CNCj 开始加工物料的时间差	秒
$\boldsymbol{\eta}$	系统的作业效率	-

注：这里我们列出了论文中使用到的大部分符号，其它符号将会在下文中详细说明。

四、 问题分析

4.1 一道工序的物料加工问题分析

对于一道工序的物料加工作业问题，由于任一台 CNC 都可以完成加工作业，因此我们的调度模型主要分析了 RGV 在多台 CNC 都发出请求指令时的调度策略：“先请求先服务（FIFO）”与“就近原则”。如果把等待执行上下料操作的多台 CNC 比做一

个队列，那么“先请求先服务”就是 RGV 优先前往指令发送时间最早、等待时间最长的 CNC 处；而“就近原则”则是指 RGV 根据 CNC 的距离远近与作业时间长短确定作业次序。

从一般情况来看，基于以上两种规则的调度算法都有不错的表现，但为了求得最优的调度策略，我们可以利用遗传算法进行优化。如果把每一台 CNC 的每一次加工都看成一个节点，那么最优调度策略必然是遍历完所有节点一次且花费总时间最少的路径。这样的话，我们的目标就变成了应用遗传算法解决一个旅行商问题：首先我们初始化一个种群，并以每个个体的染色体作为对应路径；接下来我们选择一些适应度高（路径花费总时间小）的个体以基因复制、交叉、变异等操作进行繁殖进行繁殖；最后经过大量的迭代后，即可求得最优的遍历路径。

4.2 两道工序的物料加工问题分析

对于两道工序的物料加工作业问题，我们需要首先确定 CNC 的最优组合方式，即哪些执行第一道工序，哪些执行第二道工序，且此时能加工完成的零件数 n 达到最大。因此我们考虑穷举所有可能的 CNC 组合方式，并得到了此时的最优方案与 n。与第一种情况类似，接下来我们利用离散化粒子群算法求解 RGV 的最优调度策略：随机初始化一个粒子集群，并以其对应路径的总时间的倒数作为适应度值；利用速度迭代方程与位置迭代方程不断更新粒子速度，再记录下每个粒子的局部最优解与集群的全局最优解；当迭代次数达到设定值时，停止迭代，此时的全局最优解即为最优的调度策略。

4.3 CNC 的故障问题分析

由于 CNC 在物料加工过程中故障发生的随机性，我们将无法再利用遗传算法或者离散粒子群算法求解全局的最优调度策略。因此，在这里我们主要考虑 RGV 在机器发生故障时的响应机制：“就近原则”，即 RGV 会移动到距离故障机器最近的其它机器处作业。此外，我们还基于蒙特卡洛算法对该调度策略进行了多次仿真以评估调度算法的性能。

五、 一道工序下的物料加工 RGV 动态调度模型

5.1 模型的建立与求解

5.1.1 先请求先服务与就近原则调度算法

显然，当物料的加工仅需要进行一道工序时，为了最大化机器的利用率，我们可以给每台 CNC 都安装上同样的刀具，即任意一台 CNC 都可以完成物料的加工作业。事实上，对于系统的整个加工作业流程，我们可以将其大致分为以下几个部分：

- 最开始时，所有 CNC 发出上料请求指令，RGV 接收指令并自行确定其上料顺序。
- RGV 移动到相应的 CNC 面前，执行上料操作。
- CNC 开始对物料进行加工，而 RGV 移动至其他 CNC 处再次进行上料操作。
- 当 CNC 加工完成后发出上下料请求，RGV 再接收指令并移动到对应的 CNC 前完成上下料与清洗操作，如此循环往复。

在上述简单系统加工作业流程的基础上，我们需要考虑当有多台 CNC 发出需求信号时，RGV 的调度策略：**“先请求先服务”**或**“就近原则”**。“先请求先服务”（FIFO,First In First Out）指的是 RGV 按照 CNC 发出需求信号的时间先后确定其作业次序，即 RGV 优先选择等待时间长的 CNC 进行操作；“就近原则”则是指 RGV 根据 CNC 的距离与作业时间确定作业次序，即优先距离最近，其次作业时间最短。

实际上，“先请求先服务”与“就近原则”分别体现了两种算法思想，前者模拟的是正在排队等待某项操作的**队列**，队伍里越前的人，其操作越先得到执行，他也就越先离开；而后者体现的则是贪心思想，在每一次调度时，我们都力求得到**局部最优解**：即先给最近的 CNC 上下料。

5.1.2 基于遗传算法的优化调度模型

基于上述两种简单的调度算法，我们能够分析得到智能加工系统在一个班次内的物料加工数量 n。但显然，由于该算法的简易性，得到的 RGV 调度策略也不一定是最优的，接下来我们考虑将动态调度问题转化为相应的 TSP（旅行商）问题，并利用遗传算法对其求解 [1]。

首先，我们将每一台 CNC 的每一次加工过程都看成一个节点，那么在一个班次内，这样的节点共有 n 个。因此，我们的目标就是寻找到一条最优路径，它能**让** RGV **遍历完所有的节点一次且付出的时间代价最小**，而这恰好就是一个经典的旅行商问题。

我们先来讨论一个简单的例子，假设我们已知每台 CNC 只加工了一个物料，那么此时 n 为 8，即 8 个节点（我们将这 8 个节点标号为 1 至 8）。为了更好地说明，我们现将 8 个节点及 RGV 位置的说明图绘制如下：

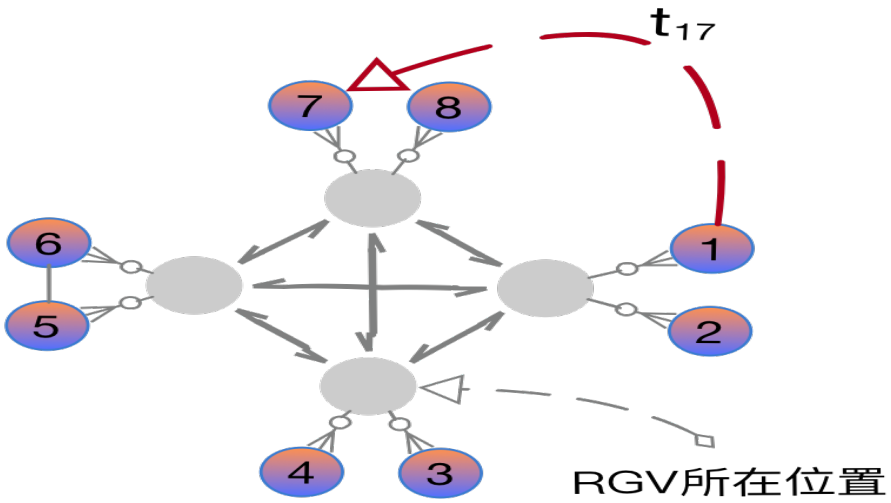

图 1　8 个节点及 RGV 所在位置的说明图

在图 1 中，红色圆圈代表了这 8 个节点，而灰色圆圈则为 RGV 的可能位置。同时，这八个节点是互相连通的，也就是说，RGV 可以从任意一个节点出发去其它任一个节点进行作业。此外，我们以 CNCi 开始加工物料到 CNCj 开始加工物料的时间差 t_{ij} 作为节点 i 到节点 j 路径上的时间代价，那么 RGV 的最优调度策略就是找到一条遍历完 8 个节点且花费总代价最小的路径。对于每条边上的 t_{ij}，我们可以表示如下：

$$t_{ij} = t_m + t_u + t_w + \Phi(x)t_c \tag{1}$$

其中，t_m 是 RGV 从 CNCi 移动到 CNCj 的花费时间，而 RGV 可能移动了 1 个单位、2 个单位或者 3 个单位；t_u 是 RGV 为 CNC 上下料一次所花费的时间；t_w 是 RGV 可能出现的等待时间，当 CNCj 处于空闲状态时，等待时间就变为了 0；t_c 则是 RGV 可能出现的清理时间，但当 CNC 是第一次上料时，清洗时间为 0，且函数 $\Phi(x)$ 定义如下：

$$\Phi(x) = \begin{cases} 0 & \text{CNC 是第一次上料} \\ 1 & \text{CNC 不是第一次上料} \end{cases} \tag{2}$$

参照图论知识，我们可以以图 1 为基础，写出其邻接代价矩阵 T，且 RGV 每遍历完一个节点后，该矩阵数值都会进行更新：

$$T = (t_{ij})_{8\times 8} = \begin{bmatrix} t_{11} & \cdots & t_{18} \\ \vdots & \ddots & \vdots \\ t_{81} & \cdots & t_{88} \end{bmatrix} \tag{3}$$

在上述讨论中，我们只考虑了短时间内每台 CNC 都加工完成一件物料的情况，而在实际的生产班次中，每台 CNC 所加工完成的物料数量 m_i 会是很多的。同时，**越高效的调度策略，每台 CNC 加工完成的物料数会越接近**，因为只有这样 RGV 的等待时

间才会越小。那么，对于最优的调度策略，我们可以认为每台 CNC 的 m_i 是完全一样的（均为 m），而这也就意味着 RGV 的调度很可能存在某种循环。

当每台 CNC 加工完成的物料数 m 大于 1 时，我们可以在之前二维连接图的旅行商问题上再增加一个 m 维度，那么其在三维空间中可以表示如下：

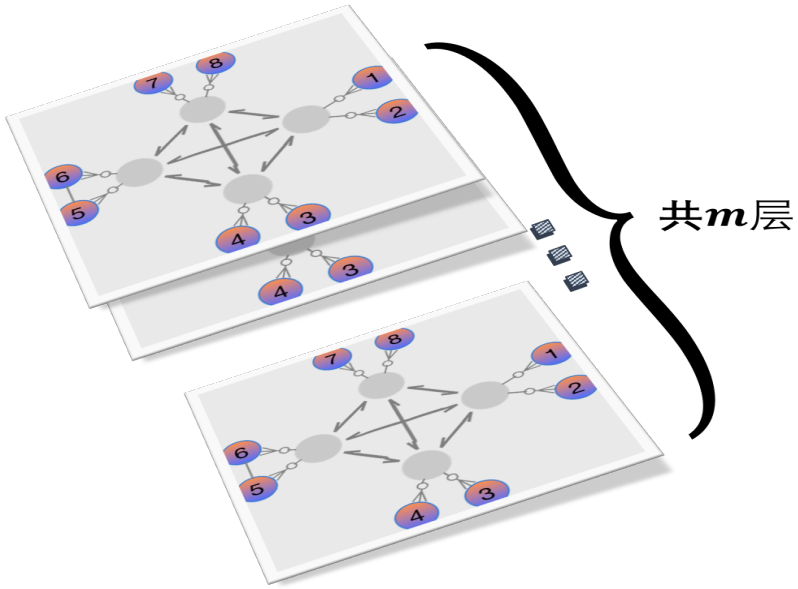

图 2　8 个节点及 RGV 在三维空间中的表示

在这里，我们需要特别说明，图 2 中**任意两点间的时间代价并不与它们的欧氏距离成正比关系**。且在同一水平面上的节点 i 与 j 之间的时间代价为 t_{ij}，不在同一水平面上的节点 i 与节点 j 的时间代价也为 t_{ij}，因为所有的 j 在 m 维度上都是等价的。

同样地，我们也需要在这个连接图中找到一条可以遍历完所有节点且总时间代价最小的最优路径。在这里，我们考虑利用遗传算法进行求解，它是借鉴生物进化过程而提出的一种启发式搜索算法。对于节点 1 到 8，我们用二进制对其编码为 000,001$\cdots$111，那么每一条路径都可以表示为一个由 0 与 1 组成的序列，即染色体。而对于每一条染色体，我们用相应路径上时间代价总和的倒数作为其适应度函数值。若路径花费的总时间代价**越小**，其对应染色体的适应度函数值就会**越高**，也就表明它越容易被选择来进行**进化**。

综上，我们可以将基于遗传算法的优化调度模型描述如下 [2]：

- 随机生成第一代种群，计算所有个体的染色体适应度函数值，即染色体对应的遍历路径的时间代价总和。
- 以染色体适应度函数值的高低为基础，选择一些个体进行繁殖。在繁殖的过程中，会发生交叉、变异等现象，这样适应度低的个体会被逐步淘汰，而适应度高的个体会越来越多。
- 当种群进行多次进化后，保存下来的个体都是适应度很高的，其中就很可能包含史

上适应度最高的个体，此时就寻找到了时间代价最小的最优路径。

同时，遗传算法的流程图为：

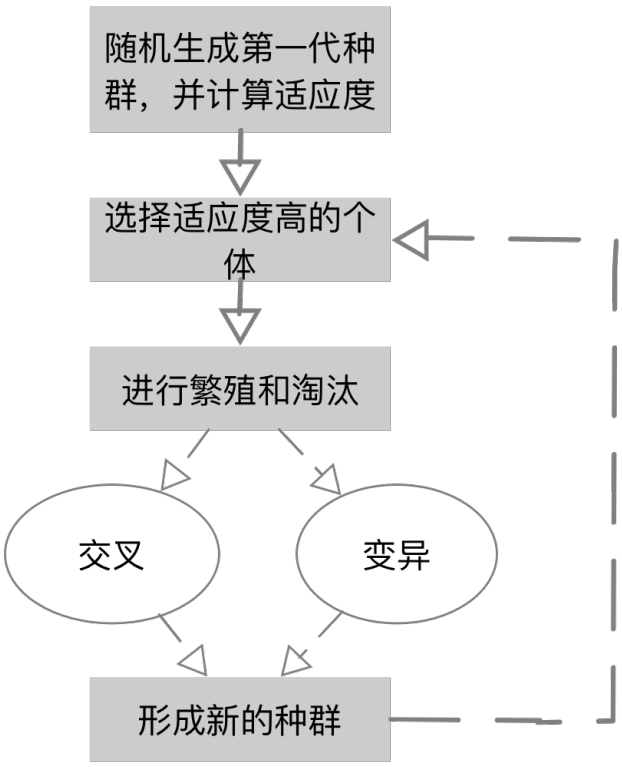

图 3　遗传算法的流程图

5.2 模型实用性与算法有效性检验

5.2.1 RGV 的调度策略

以题目所给的第一组系统作业参数值为例，我们分别以“先请求先服务”与“就近原则”调度算法为基础，编写相应代码（见附录）模拟了系统的整个作业流程：最终我们发现无论是按照“先请求先服务”原则还是“就近原则”，RGV 的调度策略都会衍变成从 CNC1 到 CNC8 的顺序执行，并不断循环下去。其实这也很好理解，观察表中的作业参数，我们会发现 CNC 加工完成一个一道工序的物料所需时间很长，这也就意味着当 RGV 完成所有当前作业任务时，总是会出现**原地等待**的情况，因为此时 CNC 仍处于物料加工状态，当 CNC 加工完成后 RGV 再根据相应原则确定其上下料顺序。最终 RGV 的调度策略也就会跟最开始的上料次序一样：从 CNC1 到 CNC8 顺序执行，并一直循环下去了。

在上述两种调度策略的基础上，我们再利用遗传算法进行优化求解（代码见附录）：首先随机生成一些路径，即第一代的种群，并以时间代价总和的倒数为适应度函数计算每个个体的适应度；采用轮盘赌选择方法（各个个体被选中的概率与其适应度大小成正

比）选择一些个体进行繁殖，同时设置交叉概率为 0.65，变异概率为 0.01；迭代 4000 次后我们得到了最优个体，其所表示的正是 RGV 从节点 1 顺序遍历到节点 8 再不断循环的路径；此外，最优适应度为 $\frac{1}{1621}$。下图展示了遗传算法调度结果的甘特图，其中横坐标为时间，纵坐标为 CNC 的编号，不同的颜色代表着不同的机器正处于忙碌的加工状态，空白则表示为空闲状态。

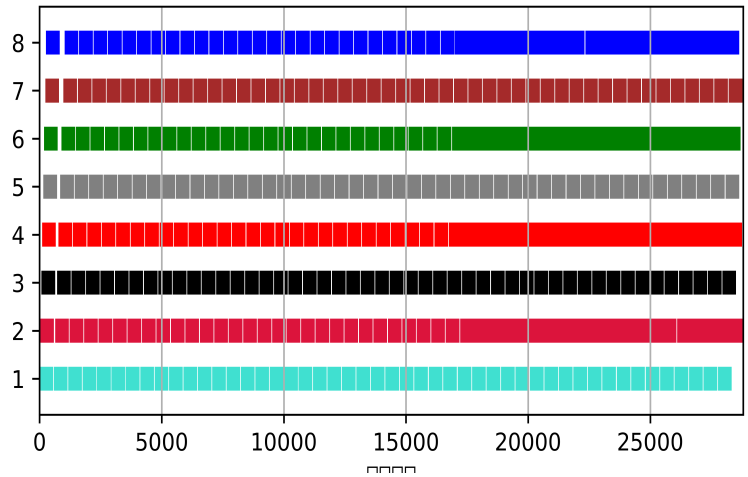

图 4　遗传算法调度结果的甘特图

从甘特图中我们可以看出，CNC 机器绝大多数时间都处于忙碌的物料加工状态，系统整体运行效率很高。

同理，对于第二组参数与第三组参数，我们也得到了其调度策略，且均为 RGV 从 CNC1 到 CNC8 顺序执行的循环。（RGV 的具体调度策略见表格 $case1\_res$）

5.2.2 系统的作业效率

在这里，我们以每台 CNC 的平均加工时间与一轮班次的总时间之比作为系统的作业效率 η，那么 η 可以表示如下：

$$\eta = \frac{\text{完成的物料加工数量} \times \text{加工时间}}{\text{CNC 数量} \times \text{一轮班次的总时间}} = \frac{n \times t_p}{8 \times 8 \times 3600} \tag{4}$$

分别将每一组作业的物料加工时间 t_p （560,580,545）代入式 4 中，我们可以得到第一组数据的作业效率为：**93.1**%；第二组数据的作业效率为：**90.6**%；第三组数据的作业效率为：**92.7**%。我们发现三组数据的作业效率都很高且超过了百分之 90，这也恰好说明了我们的调度算法的优越性。

六、 两道工序下的物料加工 RGV 动态调度模型

6.1 基于离散粒子群优化算法的动态调度模型

当物料加工作业需要由两道工序完成时，它的第一道与第二道工序要分别由两台不同的 CNC 执行，而这也就意味着我们需要首先分析哪些 CNC 负责加工第一道工序，哪些负责第二道，以及在此基础上 RGV 的最优调度策略，以加工完成更多的物料 [3]。

为了找到加工第一道工序与第二道工序的 CNC 数量以及其组合方式，我们首先遍历了所有 256（2^8）种可能情况，并在每种情况下利用“就近原则”（即当多台 CNC 响应时，无论其执行的是第一道工序还是第二道工序，RGV 都会优先前往距离最近、上下料时间最短的 CNC 处）分析其一个班次内的物料加工数量 n，最终得到了在 n 值最大的情况下 CNC 的组合方式。

在已知 CNC 组合方式与物料加工数 n 的情况下，我们同样地将其转化成了旅行商问题，并利用离散化的粒子群算法进行优化验证 [4]。

与遗传算法类似，粒子群算法（PSO）也是一种基于生物进化方式的启发式算法。简单来说，它模拟的就是一种鸟类的觅食行为，即鸟类在不知道食物具体位置的情况下，按照一种合作的方式进行寻找，且群体中的每个成员通过学习它自身的经验和其他成员的经验来不断地改变搜索的方向，最终完成整个集群向食物的聚集。

对于集群中的每个粒子 i，我们分别用位置向量 $P_i = (p_{i1}, p_{i2} \cdots p_{it})$ 与速度向量 $V_i = (v_{i1}, v_{i2} \cdots v_{it})$ 进行描述，其中 t 为优化问题的维度大小。接下来，我们需要通过迭代来不断更新粒子的速度与位置：

$$V_i = c_1 r_1 (Pbest_i - P_i) + c_2 r_2 (Gbest - P_i) + wV_i \tag{5}$$

$$P_i = V_i + P_i \tag{6}$$

其中 c_i、c_2 是学习参数，r_1、r_2 是介于 [0,1] 之间的随机概率值，$Pbest_i$ 是粒子 i 搜索到的局部最优位置，$Gbest$ 是集群搜索到的全局最优位置，w 则是惯性权值。

为了更好地理解，我们可以把每只鸟看成所有节点的访问路径，把食物当做效率最高的最优路径，再把距离看成路径花费的总时间长短，最后维度大小就是物料加工数的两倍 $2n$（每个物料需要经过两次加工）。

但是，由于传统的粒子群算法是用于解决**连续问题**的，而在我们的调度问题中，CNC 的编号是**离散**的（从 1 到 8），因此我们考虑将其优化成离散粒子群算法（DPSO）。首先，我们定义了粒子的位置为 $2n$ 个加工作业的一个排列，粒子的速度为排列中作业的交换，并根据离散量运算的特点对粒子的运动规则进行了重定义。接下来我们对离散粒子群算法做一个描述：

- **粒子的位置向量 P_i**

  粒子的位置向量 P_i 就是 $2n$ 个作业的一种排列，它同时也代表 RGV 的一种调度方案。

- **粒子的速度向量 V_i**

  速度向量 V_i 用于改变粒子 i 的位置向量，且其每一个维度上的分量都代表着两种操作：当 v_{ij} 等于 0 时，它表示当该速度分类作用于相应粒子的位置分量 p_{ij} 上时，p_{ij} 的值并不会发生改变；而当 v_{ij} 不等于 0 时，该速度分量则会将位置分量 p_{ij} 更新为 v_{ij}。这样就保证了位置向量 P_i 在任意速度向量 V_i 的作用下依然是 $2n$ 个作业的一个排列，也保证了路径的可行性。

- **P_i 与 V_i 的加法运算**

  粒子 i 的位置移动可以由 P_i 与 V_i 的加法运算实现，即 $P_i = V_i + P_i$，那么其新位置的每一个分量 p_{ij} 均可表示为：

$$p_{ij} = \begin{cases} p_{ij} & v_{ij} = 0 \\ v_{ij} & v_{ij} \neq 0 \end{cases} \tag{7}$$

- **P_i 与 P_j 的减法运算**

  如果我们将两个位置向量 P_i 与 P_j 做减法运算，那么得到的结果其实是一个速度向量 V，即 $V = P_i - P_j$，它代表着位置 P_i 可由位置 P_j 加上速度 V 得到。对于具体的计算过程，我们可以比较 P_i 与 P_j 的每一个分量 p_{ik} 与 p_{jk}，若它们相等，则 v_k 为 0，否则 v_k 为 p_{ik}。

$$v_k = \begin{cases} 0 & p_{ik} = p_{jk} \\ p_{ik} & p_{ik} \neq p_{jk} \end{cases} \tag{8}$$

- **V_i 的概率运算**

  与传统粒子群算法中的随机概率一样，这里我们也为速度向量 V_i 设置了概率运算：$V_j = gV_i$。其中 g 就是介于 0 到 1 之间的概率值，当我们开始计算时，对于 V_i 的每一个分量 v_{ik} 都会先产生一个随机值 c，如若 c 大于 g，则 v_{jk} 等于 v_{ik}；反之 v_{jk} 为 0。

$$v_{jk} = \begin{cases} v_{ik} & c > g \\ 0 & c \leq g \end{cases} \tag{9}$$

- **粒子的速度迭代方程**

  在离散化的粒子群算法中，我们重新定义了粒子的运动规则，此时粒子的位置更新是通过速度向量一步到位的，因此取消了式（5）中的惯性权值。

$$V_i = g_1(Pbest_i - P_i) + g_2(Gbest - P_i) \tag{10}$$

- **粒子的位置迭代方程**

接下来我们再利用速度向量对粒子 i 的位置进行更新：

$$P_i = V_i + P_i \tag{11}$$

随着迭代的不断深入，粒子会不断向最优路径聚集，此时的 $Gbest$ 即可认为是最优路径。

同时，离散粒子群优化算法的流程图可描述如下 [5]：

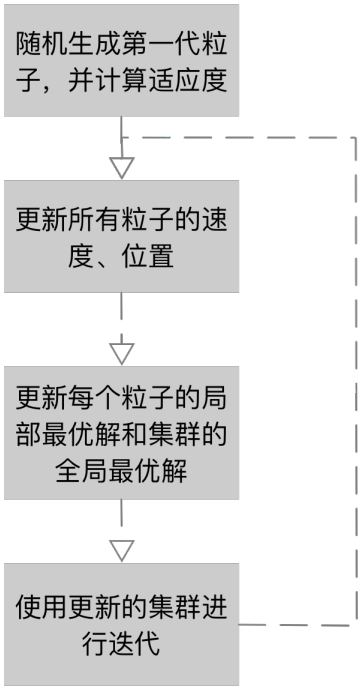

图 5　离散粒子群算法的流程图

6.2 模型实用性与算法有效性检验

6.2.1 RGV 的调度策略

以题目所给的第一组系统作业参数值为例，我们穷举了所有可能的 CNC 组合方式，并得到了在一轮班次内加工完成的物料数 n 最大下的最优组合，**即** CNC1,3,5,7 **执行第一道工序，**CNC2,4,6,8 **执行第二道工序**，且 n 等于 253。其实这也是很好理解的，因为当负责每一道工序的 CNC 数量都是 4 时，只有四个负责一道工序和四个负责二道工序的各在一排才能最大化地节省 RGV 的移动时间，以达到最优，且加工时长更大的 CNC 在上下料时间更短的那一侧。同时，我们再利用离散化的粒子群优化算法进行了 RGV 的调度策略分析（代码见附录）：随机初始化一个集群并计算每个粒子的适应度（路径总时间的倒数）；设置最大迭代次数为 1000；分别利用式（10）与式（11）不断更新粒子的速度与位置；再更新每个粒子的局部最优解 $Pbest_i$ 和集群的全局最优解 $Gbest$；当迭代次数达到 1000 时，停止迭代并输出此时的 $Gbest$。

同时，我们绘制出了在最优调度策略下的甘特图：

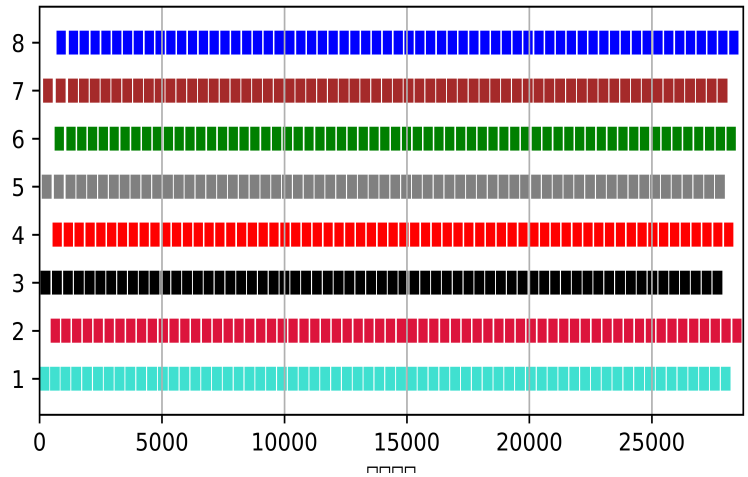

图 6　粒子群算法调度结果的甘特图

同理，第二组数据的 n 为 211，**最优组合为** CNC2,4,6,8 **执行第一道工序，**CNC1,3,5,7 **执行第二道工序**；第三组数据的 n 为 241，**最优组合为** CNC1,2,4,6,7 **执行第一道工序，**CNC3,5,8 **执行第二道工序**，RGV 的具体调度策略见表格 $case2\_res$。对于第三组数据，我们发现一个很有意思的现象：负责加工第一道工序的 CNC 有 5 台，而负责加工第二道工序的只有 3 台。实际上这是因为第一道工序的加工时间远大于第二道工序的加工时间，所以只有当负责第一道工序的 CNC 数量略大于负责第二道工序的 CNC 时才能保证系统效率最高，否则负责第二道工序的 CNC 很容易出现空闲状况。

6.2.2 系统的作业效率

与式（4）不同的是，在这里我们的加工时间为第一道工序的加工时间和第二道工序的加工时间之和，即作业效率 η 可以表示如下：

$$\eta = \frac{\text{完成的物料加工数量} \times \text{加工时间}}{\text{CNC 数量} \times \text{一轮班次的总时间}} = \frac{n \times (t_{e1} + t_{e2})}{8 \times 8 \times 3600} \tag{12}$$

分别将每一组作业的物料加工时间 t_p （778,780,637）代入式 4 中，我们可以得到第一组数据的作业效率为：**85.4**%；第二组数据的作业效率为：**71.4**%；第三组数据的作业效率为：**66.6**%。与一道工序下的物料加工相对比，我们发现两道工序时系统作业效率低了很多。这主要是因为负责第二道工序的机器只能等待物料加工完第一道工序后再执行，且两道工序的相差时间越大，系统的作业效率 η 越低。

七、 CNC 故障下的 RGV 动态调度模型

7.1 模型的建立

现在我们考虑 CNC 出现故障的情况：每台 CNC 从加工第一个物料开始运转，一直到故障发生（故障发生概率为 1%）或者一轮班次结束，且 CNC 只有在加工过程中才可能发生故障，每次故障的排除时间服从于 [600 秒,1200 秒] 的均匀分布，此时 CNC 再重新加入作业序列。

由于 CNC 故障发生的随机性，我们没有办法再将调度问题转化为相应的旅行商问题并利用遗传算法或者粒子群算法进行最优路径的求解。因此，在这里我们主要考虑的是 RGV 在机器故障下的响应机制 [6]。那么，对于一道工序的物料加工作业来说，当任一台 CNC 发生故障时，RGV 会移动到距离故障机器最近的一台 CNC 面前进行作业；而对于两道工序的物料加工作业来说，RGV 会移动到距离故障机器最近的一台负责第一道加工工序的 CNC 面前进行作业，因为无论是负责第一道工序的 CNC 发生故障亦或是负责第二道工序的 CNC 发生故障，物料的加工作业都得从第一道工序重新开始 [7]。

蒙特卡洛算法是一种以概率统计理论为基础的非常重要的统计模拟方法。它是通过把所求的问题与一定的概率模型相联系，用计算机实现统计模拟或抽样，以获得问题的近似解 [8]。因此在这里我们考虑在一道工序与两道工序的物料加工作业情况下，利用蒙特卡洛算法对 CNC 发生故障时的系统作业流程进行多次模拟，并通过系统在一个班次内的加工完成的平均物料数评估了我们调度算法的性能。

7.2 模型求解

以上述 RGV 在机器故障下的响应机制为基础，我们通过程序（见附录）对系统的作业流程进行了模拟（以第一组作业参数为例），得到了一道工序与两道工序下的甘特图：

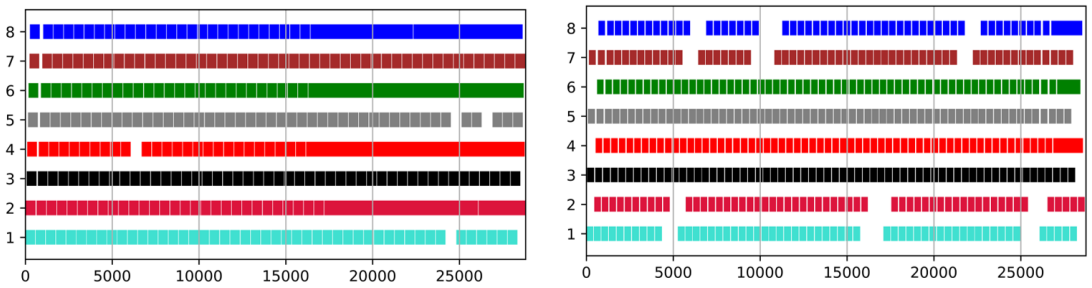

图 7　一道工序（左）与两道工序（右）下的甘特图

7.3 模型评估

正是由于故障发生的随机性，我们决定运用蒙特卡洛算法来评价调度策略的性能。由此，对于一道工序下的每一组数据，我们均模拟了 500 次系统的作业流程，并计算其在一个轮次内完成的物料加工数 n。

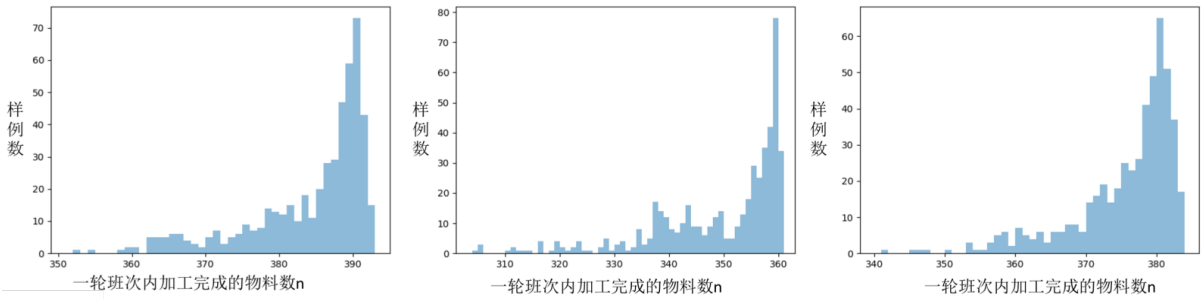

图 8　第一组数据（左）、第二组数据（中）与第三组数据（右）下的作业结果

同样地，对于两道工序下的每一组数据，我们也可以得到如下的结果：

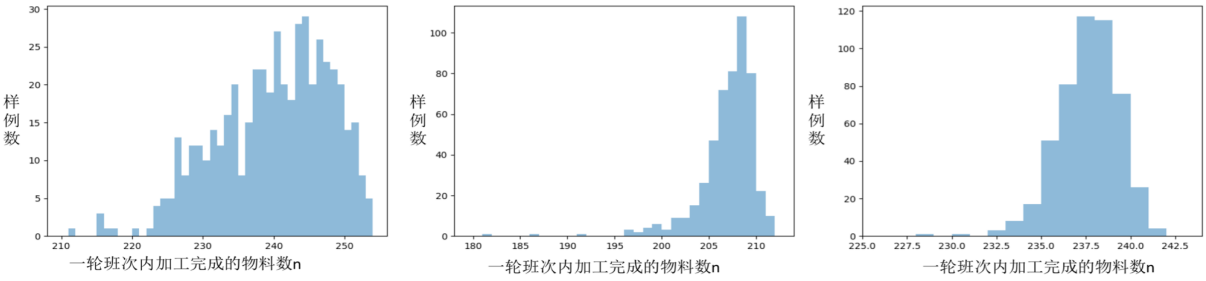

图 9　第一组数据（左）、第二组数据（中）与第三组数据（右）下的作业结果

观察以上两图，我们发现在一道工序的物料加工作业情况下，每组数据 n 的频数分布大致一样；而在两道工序时，不同组数据下 n 的频数分布则相差较大。下表为具体 n 值方差：

表 2　不同工序数和系统作业参数对 n 值方差的影响

物料的加工作业情况	第一组数据	第二组数据	第三组数据
一道工序的物料加工作业	43.41	121.53	71.64
两道工序的物料加工作业	62.95	6.36	8.68

综合分析以上图表，我们可以得到如下的结论：**当物料加工所需的工序数越多且不同工序的加工时间越相近时，其受机器故障的影响也就越大。**

7.4 系统的作业效率

在一道工序的物料加工作业情况下，我们利用式（4）计算了系统的作业效率 η。其中第一组数据的作业效率为：**92.4**%；第二组数据的作业效率为：**90.1**% ；第三组数据的作业效率为：**91.8**% 。

而在两道工序的物料加工作业情况下，我们利用式（12）计算了系统的作业效率 η。其中第一组数据的作业效率为：**81.0**%；第二组数据的作业效率为：**69.4**% ；第三组数据的作业效率为：**65.2**% 。

同时，我们发现在机器发生故障的情况下，虽然系统的作业效率有所降低，但降低地并不多。

八、 模型拓展

8.1 多个 RGV 的动态调度模型

在规模更大的生产线上，往往有更多的 CNC 需要进行资源的调配。那么一辆 RGV 小车的处理能力是远远不够的，这时候就需要加入更多的 RGV 小车来提高生产线的装配效率 [9]。但是引入更多的 RGV 可能会带来相碰撞的问题，产生这种情况的主要原因有：1. **两辆 RGV 竞争同一段线路**；2. **线路交叉**。为了解决这种问题，我们采用了时间窗（*timewindow*）调度算法 [10]。

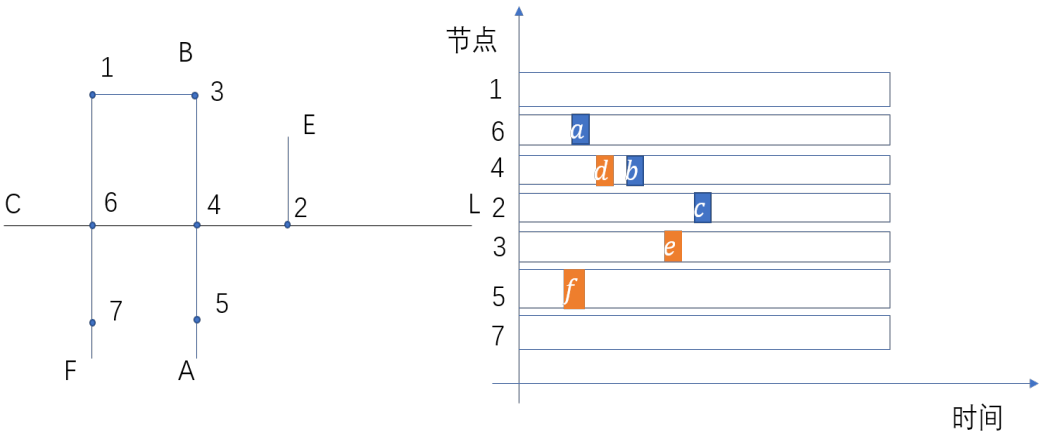

图 10　多 RGV 调度环境及其时间窗模型

如图 10 所示，在比较复杂的线路中一号 RGV 会先后经过节点 6、4、2 到达节点 L ，相应的节点**保留时间窗**为 a,b,c ；而二号 RGV 会从点 A 经过 4、3 到达 B，则相应的节点保留时间窗为 f,d,e。

如若现在有一个三号 RGV 要从 E 走到 F 点，我们则需要对其规划一条与 RGV1 和 RGV2 没有时间窗冲突且物理上可达的路径。即在多 RGV 系统中，从一点到另一

点不发生碰撞的条件是同时满足**空间可行性**（两地之间存在物理通路）与**时间可行性**（RGV 在自由时间窗内能够到达目的地）。

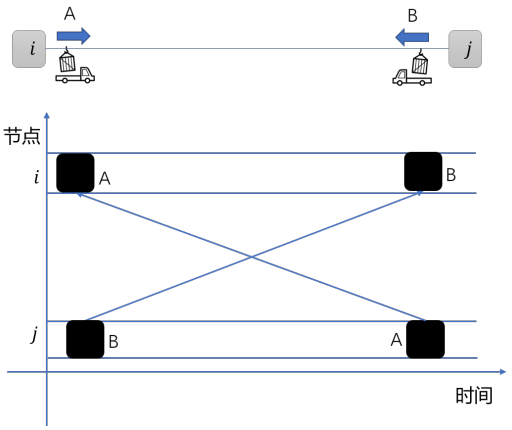

图 11　碰撞进行 1 及其时间窗

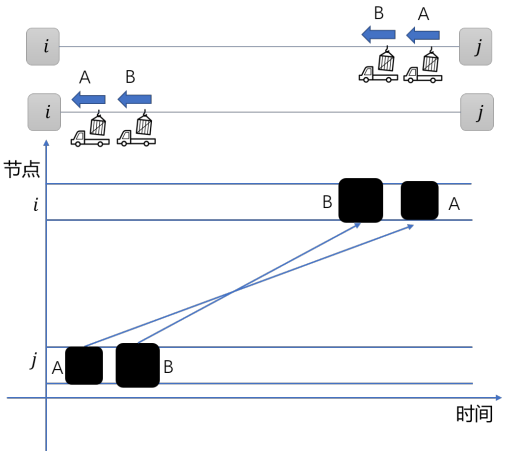

图 12　碰撞进行 2 及其时间窗

将以上模型应用于直线道路中，我们可以发现碰撞的情形可以归纳为以下两种：如上左图所示，若两个节点占用时间窗的顺序相反，则一定会发生碰撞。再如上右图所示，若发生了上面左图的情况，那么 RGVA 会先到达 j，离开后 RGVB 才可以到达；但如果 RGVA 突然减速或者停下，此时 RGVA 占用此保留时间窗的时间就会**延长**；若这时 RGVB 仍按照原计划前进，两者则会发生碰撞。在对多 RGV 加工系统的调度进行规划时，我们应当特别注意以上两种情况。

九、 模型评价

9.1 模型的优缺点

9.1.1 模型的优点

- **调度策略的优化**
  对于一道工序与两道工序的物料加工作业情况，我们先分别考虑了在“先请求先服务”与“就近原则”调度算法下的调度策略，并利用遗传算法、离散粒子群算法等启发式搜索算法进行了调度策略的优化。
- **将调度决策转化为了旅行商问题**
  我们把每台 CNC 的每次加工过程都看成了一个节点，将 RGV 的最优调度策略转化为了寻找遍历完所有节点一次且花费时间最小的最优路径，由此将动态调度问题转化为了一个旅行商问题。
- **粒子群算法的离散化**
  考虑到传统的粒子群算法解决的是连续问题，而调度问题中 CNC 的编号是离散的，因此我们重新定义了粒子的速度与运算规则，以离散化粒子群算法。

9.1.2 模型的缺点

- **算法迭代速度较慢**
  其实无论是遗传算法还是粒子群算法，它们的迭代速度都比较慢，而且易于陷入局部最优解。
- **简化了机器故障的分析**
  在机器故障时，我们仅考虑了 RGV 此时的响应机制并将其简化为了“就近原则”，这在一定程度上忽略了其它因素带来的影响。
- **机器故障下的调度策略并不一定是最优解**
  我们在机器故障的情况下以“就近原则”得到的调度策略实际上是近优解，而非最优解。

9.2 模型的改进方向

- 对遗传算法及粒子群算法进一步优化，使其跳过局部最优解，更快地找到全局最优解。
- 编写代码对多 RGV 加工系统的作业流程进行仿真。

参考文献

[1] 吴焱明, 刘永强, 张栋, 等. 基于遗传算法的 RGV 动态调度研究 [J]. 起重运输机械, 2012(6):20-23.

[2] http://blog.jobbole.com/110913/

[3] 蒋维, 陈开, 钟小强, 等. 基于动态规划的资源受限随机工序调度 [J]. 计算机工程, 2008, 34(16):19-21.

[4] 虞斌能, 焦斌, 顾幸生. 改进协同粒子群优化算法及其在 FlowShop 调度中的应用 [J]. 华东理工大学学报, 2009, 35(3):468-474.

[5] https://www.cnblogs.com/maybe2030/p/5043356.html

[6] 李素粉, 朱云龙, 尹朝万. 具有随机加工时间和机器故障的流水车间调度 [J]. 计算机集成制造系统, 2005, 11(10):1425-1429.

[7] 周强. 考虑随机加工时间和机器故障的多目标流水车间调度 [J]. 计算机时代, 2008(6):9-10.

[8] https://www.cnblogs.com/ECJTUACM-873284962/p/6892022.html

[9] 贺丽娜, 楼佩煌, 钱晓明, 等. 基于时间窗的自动导引车无碰撞路径规划 [J]. 计算机集成制造系统, 2010, 16(12):2630-2634.

[10] 孙亮. AGVS 中避碰问题的研究现状与发展趋势 [J]. 物流技术, 2005(3):25-27.

附录 A 算法源程序

```
case1 = [383, 360, 392]
case2 = [253, 211, 241]
case3_1 = [380, 358, 388]
case3_2 = [240, 205, 236]

cost = [560, 580, 545]
cost_2 = [400 + 378, 280 + 500, 455 + 182]
case1_cost = [ ]
case2_cost = [ ]
case3_1_cost = [ ]
case3_2_cost = [ ]
for i in range(3):
case1_cost.append(case1[i] * cost[i])
case2_cost.append(case2[i] * cost_2[i])
case3_1_cost.append(case3_1[i] * cost[i])
case3_2_cost.append(case3_2[i] * cost_2[i])

for i in range(3):
case1_cost[i] /= 28800
case2_cost[i] /= 28800
case3_1_cost[i] /= 28800
case3_2_cost[i] /= 28800
for i in range(3):
case1_cost[i] /= 8
case2_cost[i] /= 8
case3_1_cost[i] /= 8
case3_2_cost[i] /= 8
```

```
class CNC(object):
def __init__(self, CNC_type=0, time_cost=(0, 0), ID=0):
self.type = CNC_type

if self.type == 0:
self.do_first_time = time_cost[0]
else:
self.do_first_time = time_cost[0]
self.do_sec_time = time_cost[1]

self.ID = ID
self.locate = ID // 2
self.end_time = -1
self.list_put_time = []
self.list_fetch_time = []
self.list_index = []
```

```
self.is_break = False
self.work_on_index = -1

self.is_last_break = False

def get_pram(self):
dict = {'ID': self.ID, 'CNC_type': self.type}
if self.type == 0:
dict['time_cost'] = (self.do_first_time,)
else:
dict['time_cost'] = (self.do_first_time, self.do_sec_time)
return dict

def do_process(self, begin_time, type=0):
if self.query_end_time() <= begin_time:
if type == 0:
self.end_time = begin_time + self.do_first_time
if type == 1:
self.end_time = begin_time + self.do_sec_time

def query_end_time(self):
return self.end_time

def update_end_time(self, now_time, process_type=0):
if process_type == 0:
self.end_time = now_time + self.do_first_time
else:
self.end_time = now_time + self.do_sec_time

def record_put_time(self, put_time):
if self.ID == 1:
a = 1 + 1
pass
if len(self.list_put_time) == 0:
self.list_put_time.append(put_time)
return

if len(self.list_put_time) == len(self.list_fetch_time):
self.list_put_time.append(put_time)
else:
self.list_put_time.append(put_time)
self.list_fetch_time.append(self.list_put_time[-1])

def record_index(self, index):
self.work_on_index = index
self.list_index.append(index)
```

```
def del_last_put_time(self):
self.list_put_time.pop(-1)

def now_break(self, break_end_time):
self.end_time = break_end_time
self.is_break = True
self.list_fetch_time.append(-1)

def recover(self, now_time):
self.end_time = now_time

def produce_fetch_time(self):
# self.list_fetch_time = self.list_put_time[1:]
self.list_put_time = self.list_put_time[:-1]
self.list_index = self.list_index[:-1]
pass
```

```
# coding=UTF-8
import pandas as pd
import io
import matplotlib.pyplot as plt

plt.rcParams['font.sans-serif'] = ['SimHei'] # 用来正常显示中文标签
plt.rcParams['axes.unicode_minus'] = False # 用来正常显示负号

height = 4 # 柱体高度，设为2的整数倍，方便Y轴label居中，如果设的过大，柱体间的间距就看不到了，需要修改下面间隔为更大的值
interval = 4 # 柱体间的间隔
colors = ("turquoise", "crimson", "black", "red", "grey", "green", "brown", "blue") # 颜色，不够再加
x_label = u"调度时刻" # 设置x轴label

df = pd.read_excel('output.xlsx')
df["diff"] = df.put_time - df.fetch_time

fig, ax = plt.subplots(figsize=(6, 3))
labels = []
count = 0
for i, machine in enumerate(df.groupby("machine_ID")):
labels.append(machine[0])
data = machine[1]
for index, row in data.iterrows():
# wait = 560
if row["machine_ID"] % 2 == 1:
wait = 400
else:
```

```
wait = 378
ax.broken_barh([(row["put_time"], wait)], ((height + interval) * i + interval, height),
facecolors=colors[i])
# plt.text(row["put_time"], (height + interval) * (i + 1), row['ID'], fontsize='x-small')
if row["fetch_time"] > count:
count = row["fetch_time"]
ax.set_ylim(0, (height + interval) * len(labels) + interval)
ax.set_xlim(0, count + 2)
ax.set_xlabel(x_label)
ax.set_yticks(range(interval + height // 2, (height + interval) * len(labels), (height + interval)))
ax.set_yticklabels(labels)
# ax.grid(True) # 显示网格
ax.xaxis.grid(True) # 只显示x轴网格
# ax.yaxis.grid(True) # 只显示y轴网格
plt.savefig('gantt.png', dpi=800)
plt.show()
```

```
from process_system import ProcessSystem
import pandas as pd
import logging
import random

from deap import base
from deap import creator
from deap import tools

data_input = {
'group_1': [20, 33, 46, 560, 400, 378, 28, 31, 25],
'group_2': [23, 41, 59, 580, 280, 500, 30, 35, 30],
'group_3': [18, 32, 46, 545, 455, 182, 27, 32, 25]
}

creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list , fitness=creator.FitnessMax)

toolbox = base.Toolbox()

# Attribute generator
#                      define 'attr_bool' to be an attribute ('gene')
#                      which corresponds to integers sampled uniformly
#                      from the range [0,1] (i.e. 0 or 1 with equal
#                      probability)
toolbox.register("attr_bool", random.randint, 0, 8)

# Structure initializers
#                         define 'individual' to be an individual
```

```
#                             consisting of 100 'attr_bool' elements ('genes')
toolbox.register("individual", tools.initRepeat, creator.Individual,
toolbox.attr_bool, 40)

# define the population to be a list of individuals
toolbox.register("population", tools.initRepeat, list , toolbox.individual)



# the goal ('fitness') function to be maximized
def evalOneMax(individual):
system = ProcessSystem(data_input['group_1'], system_type=0)
return system.run(individual),
# return sum(individual),



#  ----------
# Operator registration
#  ----------
# register the goal / fitness function
toolbox.register("evaluate", evalOneMax)

# register the crossover operator
toolbox.register("mate", tools.cxTwoPoint)

# register a mutation operator with a probability to
# flip each attribute/gene of 0.05
toolbox.register("mutate", tools.mutUniformInt, low=0, up=8, indpb=0.3)

# operator for selecting individuals for breeding the next
# generation: each individual of the current generation
# is replaced by the 'fittest' (best) of three individuals
# drawn randomly from the current generation.
toolbox.register("select", tools.selTournament, tournsize=3)



#  ----------

def main():
# random.seed(64)

# create an initial population of 300 individuals (where
# each individual is a list of integers)
pop = toolbox.population(n=300)

# CXPB is the probability with which two individuals
#         are crossed
#
```

```
# MUTPB is the probability for mutating an individual
CXPB, MUTPB = 0.5, 0.2

print("Start of evolution")

# Evaluate the entire population
fitnesses = list(map(toolbox.evaluate, pop))
for ind, fit in zip(pop, fitnesses):
ind.fitness.values = fit

print("  Evaluated %i individuals" % len(pop))

# Extracting all the fitnesses of
fits = [ind.fitness.values[0] for ind in pop]

# Variable keeping track of the number of generations
g = 0

# Begin the evolution
while max(fits) < 2300 and g < 1000:
# A new generation
g = g + 1
print("-- Generation %i --" % g)

# Select the next generation individuals
offspring = toolbox.select(pop, len(pop))
# Clone the selected individuals
offspring = list(map(toolbox.clone, offspring))

# Apply crossover and mutation on the offspring
for child1, child2 in zip(offspring[::2], offspring[1::2]):

# cross two individuals with probability CXPB
if random.random() < CXPB:
toolbox.mate(child1, child2)

# fitness values of the children
# must be recalculated later
del child1.fitness.values
del child2.fitness.values

for mutant in offspring:

# mutate an individual with probability MUTPB
if random.random() < MUTPB:
toolbox.mutate(mutant)
del mutant.fitness.values
```

```
# Evaluate the individuals with an invalid fitness
invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
fitnesses = map(toolbox.evaluate, invalid_ind)
for ind, fit in zip(invalid_ind, fitnesses):
ind.fitness.values = fit

print("  Evaluated %i individuals" % len(invalid_ind))

# The population is entirely replaced by the offspring
pop[:] = offspring

# Gather all the fitnesses in one list and print the stats
fits = [ind.fitness.values[0] for ind in pop]

length = len(pop)
mean = sum(fits) / length
sum2 = sum([x * x for x in fits])
std = abs(sum2 / length - mean ** 2) ** 0.5
best_one = tools.selBest(pop, 1)[0]
print(best_one)
print("  Min %s" % min(fits))
print("  Max %s" % max(fits))
print("  Avg %s" % mean)
print("  Std %s" % std)

print("-- End of (successful) evolution --")

best_ind = tools.selBest(pop, 1)[0]
print("Best individual is %s, %s" % (best_ind, best_ind.fitness.values))


if __name__ == "__main__":
main()

# if __name__ == '__main__':
#     # system = ProcessSystem(data_input['group_2'], system_type=1, CNC_type=(0, 1, 0, 1, 0, 1, 0, 1))
#     system = ProcessSystem(data_input['group_1'], system_type=0)
#     system.prt_pram()ç
#     seq = [2 for x in range(400)]
#     cnt = system.run(seq=seq)
#
#     # logging.basicConfig(filename='logger.log', level=logging.INFO)
#     # for i in range(0, 256):
#     #     # system = ProcessSystem(data_input['group_1'], system_type=1, CNC_type=(0, 1, 0, 1, 0, 1,
    0, 1))
#     #     cnc_type = []
```

```
#     #     for j in range(8):
#     #         cnc_type.insert(0, (i >> j) & 1)
#     #     system = ProcessSystem(data_input['group_1'], system_type=1, CNC_type=cnc_type)
#     #     # system = ProcessSystem(data_input['group_1'], system_type=0)
#     #
#     #     # system.prt_pram()
#     #     cnt = system.run()
#     #
#     #     logging.info(f'code:{cnc_type}, finish = {cnt}')
#
#     # creator.create("FitnessMax", base.Fitness, weights=(1.0,))
#     # creator.create("Individual", list , fitness=creator.FitnessMax)
```

```
from process_system import ProcessSystem
import pandas as pd
import logging
import random

from deap import base
from deap import creator
from deap import tools

data_input = {
'group_1': [20, 33, 46, 560, 400, 378, 28, 31, 25],
'group_2': [23, 41, 59, 580, 280, 500, 30, 35, 30],
'group_3': [18, 32, 46, 545, 455, 182, 27, 32, 25]
}

if __name__ == '__main__':
do_top = True
if do_top:
system = ProcessSystem(data_input['group_1'], system_type=1, CNC_type=(1, 0, 1, 0, 1, 0, 1, 0))
# system = ProcessSystem(data_input['group_2'], system_type=1, CNC_type=(1, 0, 1, 0, 1, 0, 1, 0))
# system = ProcessSystem(data_input['group_3'], system_type=1, CNC_type=(0, 0, 1, 0, 1, 0, 1, 0))
# system = ProcessSystem(data_input['group_1'], system_type=0)
system.prt_pram()
cnt = system.run()
else:
# find the best hyper-pram
logging.basicConfig(filename='logger.log', level=logging.INFO)
dict = {
'CNC': [],
'cnt_finish': [],
}
for i in range(0, 256):
# system = ProcessSystem(data_input['group_1'], system_type=1, CNC_type=(0, 1, 0, 1, 0, 1, 0, 1))
```

```
cnc_type = []
for j in range(8):
cnc_type.insert(0, (i >> j) & 1)
system = ProcessSystem(data_input['group_3'], system_type=1, CNC_type=cnc_type)
# system = ProcessSystem(data_input['group_1'], system_type=0)

# system.prt_pram()
cnt = system.run()
dict['CNC'].append(cnc_type)
dict['cnt_finish'].append(cnt)
logging.info(f'code:{cnc_type}, finish = {cnt}')

filepath = './rank.xlsx'
writer = pd.ExcelWriter(filepath)
df = pd.DataFrame(dict)
# columns参数用于指定生成的excel中列的顺序
df.to_excel(writer, columns=['CNC', 'cnt_finish'], index=False, encoding='utf-8',
sheet_name='Sheet')
writer.save()
```

```
from RGV import RGV
from CNC import CNC
import pandas as pd
import numpy as np
import random



class ProcessSystem(object):
def __init__(self, time_cost, system_type=0, CNC_type=(0, 0, 0, 0, 0, 0, 0, 0)):
# init pram

self.move_cost = time_cost[:3]
self.do_cost = time_cost[3:6]
self.upload_cost = time_cost[6:8]
self.wash_cost = time_cost[8]
self.CNC_list = []
self.system_type = system_type
self.dt = {}

self.break_item = []

for i in range(0, 8):
if self.system_type == 0:
self.CNC_list.append(CNC(CNC_type=CNC_type[i], time_cost=(self.do_cost[0],), ID=i))
else:
self.CNC_list.append(CNC(CNC_type=CNC_type[i], time_cost=self.do_cost[1:3], ID=i))
```

```
self.RGV = RGV(move_time=self.move_cost, do_cost_time=self.upload_cost, wash_time=self.wash_cost)

self.now_time = 0
self.end_time = 8 * 60 * 60

def run(self, seq=None):
step = 0
cnt_process_finish = 0
RGV_work_time = 0
work_index = 1
breaked = False
while self.now_time < self.end_time:
# begin_time to record the spare time of RGV
begin_time = self.now_time
step_cost_time = 0
list_cnc_end_time = []
for index, next_src in enumerate(self.CNC_list):
if next_src.end_time < 0:
list_cnc_end_time.append(-1)
continue
item_prepare_time = next_src.end_time - ([0, ] + self.RGV.move_time)[abs(self.RGV.loc_pos - index // 2)]
list_cnc_end_time.append(item_prepare_time)

next_ID = -1
list_first_ready = []
list_sec_ready = []

a_break_time = random.randint(10 * 60, 20 * 60)

for index, item in enumerate(self.CNC_list):
if item.is_break and item.end_time < self.now_time:
item.work_on_index = -1
self.CNC_list[index].is_break = False
self.CNC_list[index].is_last_break = True

for i, next_src in enumerate(list_cnc_end_time):
if next_src < self.now_time:
if self.CNC_list[i].type == 0:
list_first_ready.append(i)
elif self.CNC_list[i].type == 1:
list_sec_ready.append(i)
else:
# maybe machine break
pass

# 固定时间
```

```
# for break_machine_index in range(8):
#     if (break_machine_index not in list_first_ready or
#         break_machine_index not in list_sec_ready) and \
#             not self.CNC_list[break_machine_index].is_break \
#             and random.random() < 0.01:
#         self.dt[self.CNC_list[break_machine_index].list_index[-1]] = [break_machine_index,
    self.now_time,
#                                                                       self.now_time + a_break_time]
#         self.CNC_list[break_machine_index].now_break(self.now_time + a_break_time)
#         breaked = True

if not self.RGV.is_holding():
if len(list_first_ready) != 0:
# do some for the next_ID
# next_ID = list_first_ready[0]
next_ID = self.get_id_nealy(list_first_ready)
# next_ID = self.get_id_FIFO(list_first_ready)
# if seq is not None and step < len(seq) and seq[step] in list_first_ready:
#     next_ID = seq[step]
#     step = step + 1
# else:
#     next_ID = -1
elif self.system_type == 1 and len(list_sec_ready) > 0:
for next_src in list_sec_ready:
for next_dest in list_first_ready:
if self.RGV.get_go_pos_cost(next_src // 2) + self.RGV.do_upload(
next_src) + self.RGV.wash_time \
< self.CNC_list[next_dest].end_time - self.now_time + \
([0, ] + self.move_cost)[abs(next_src // 2 - next_dest // 2)]:
next_ID = next_src

if self.RGV.is_holding() and len(list_sec_ready) != 0:
# next_ID = list_sec_ready[0]
next_ID = self.get_id_nealy(list_sec_ready)
# next_ID = self.get_id_FIFO(list_sec_ready)
# if seq is not None and step < len(seq) and seq[step] in list_sec_ready:
#     next_ID = seq[step]
#     step = step + 1
# else:
#     next_ID = -1

# no CNC send signal
if next_ID == -1:
self.now_time += 1
continue

# print(f'now time: {self.now_time}')
```

```
# print(f'all  first  CNC ready to reload: {list_first_ready}')
# print(f'all  second CNC ready to reload: {list_sec_ready}')
# print(f'RGV is holding:{self.RGV.is_holding()}')
# go to CNC and do upload
go_pos_time_cost = self.RGV.do_go_pos(next_ID // 2)
# goto CNC time
step_cost_time += go_pos_time_cost
# record the put time
self.CNC_list[next_ID].record_put_time(self.now_time + step_cost_time)
# upload time
step_cost_time += self.RGV.do_upload(self.CNC_list[next_ID])

# update the CNC end time
if  self.RGV.is_holding():
self.CNC_list[next_ID].update_end_time(self.now_time + step_cost_time, process_type=1)
self.RGV.put()
# self.CNC_list[next_ID].record_index(work_index)
self.CNC_list[next_ID].record_index(self.RGV.hold_index)
# if it's not the first time, you need wash, it costs times, and finish a work
if list_cnc_end_time[next_ID] != -1:
step_cost_time += self.RGV.do_wash()
cnt_process_finish += 1
if random.random() < 0.01:
self.dt[self.CNC_list[next_ID].list_index[-1]] = [next_ID,
self.now_time,
self.now_time + a_break_time]
self.CNC_list[next_ID].now_break(self.now_time + a_break_time)
else:
if  self.system_type == 0:
self.CNC_list[next_ID].record_index(work_index)
work_index += 1
self.CNC_list[next_ID].update_end_time(self.now_time + step_cost_time, process_type=0)
if list_cnc_end_time[next_ID] != -1:
step_cost_time += self.RGV.do_wash()
cnt_process_finish += 1
if random.random() < 0.01:
self.dt[self.CNC_list[next_ID].list_index[-1]] = [next_ID,
self.now_time,
self.now_time + a_break_time]
self.CNC_list[next_ID].now_break(self.now_time + a_break_time)

else:
if  self.CNC_list[next_ID].type == 0:
if list_cnc_end_time[next_ID] != -1:
self.RGV.hold_index = self.CNC_list[next_ID].work_on_index
self.CNC_list[next_ID].record_index(work_index)
work_index += 1
```

```
self.CNC_list[next_ID].update_end_time(self.now_time + step_cost_time, process_type=0)
# if it's not the first time, you can hold the first_done
if list_cnc_end_time[next_ID] != -1:
if self.CNC_list[next_ID].is_last_break == False:
self.RGV.hold()
if random.random() < 0.01:
self.dt[self.CNC_list[next_ID].list_index[-1]] = [next_ID,
self.now_time,
self.now_time + a_break_time]
self.CNC_list[next_ID].now_break(self.now_time + a_break_time)
self.break_item.append(self.CNC_list[next_ID].work_on_index)
else:
self.CNC_list[next_ID].is_last_break = False

else:
if list_cnc_end_time[next_ID] != -1:
step_cost_time += self.RGV.do_wash()
cnt_process_finish += 1
raise Exception("do effect")

# print(f'finish:{cnt_process_finish}')
# update now time
self.now_time += step_cost_time
# print(f'finish time:{self.now_time}')
RGV_work_time += self.now_time - begin_time
# print ('----------')
# print('**********')
# print(f'finish:{cnt_process_finish}')
# print(f'RGV_free_time:{self.end_time - RGV_work_time}')
# print('**********')
print(self.dt)
print(self.break_item)
self.save_data()
return cnt_process_finish

def get_id_nealy(self, list):
distance = [abs(item // 2 - self.RGV.get_now_pos()) for item in list]
min_value = min(distance)
index = distance.index(min_value)
return list[index]

def get_id_FIFO(self, list):
time = [self.CNC_list[item].end_time for item in list]
min_value = min(time)
index = time.index(min_value)
return list[index]
```

```
def prt_pram(self):
print('RGV pram:')
print(f'move 1,2,3 step cost:{self.RGV.move_time}')
print(f'upload cost:{self.RGV.do_cost_time}')
print(f'wash cost:{self.wash_cost}')
print(' ')
print('CNC pram:')
for item in self.CNC_list:
print(item.get_pram())

def save_data(self, file_path='./output.xlsx'):
if self.system_type == 0:
dit = {
'ID': [],
'machine_ID': [],
'put_time': [],
'fetch_time': []
}
for item in self.CNC_list:
item.produce_fetch_time()
# dit = {'ID': [item.ID for _ in range(len(item.list_fetch_time))], 'put_time': item.list_put_time,
#         'fetch_time': item.list_fetch_time}
dit['machine_ID'] += [item.ID for _ in range(len(item.list_fetch_time))]
dit['ID'] += item.list_index
dit['put_time'] += item.list_put_time
dit['fetch_time'] += item.list_fetch_time
# sort
tem = zip(dit['put_time'], dit['machine_ID'], dit['ID'], dit['fetch_time'])
tem = sorted(tem)
dit['put_time'], dit['machine_ID'], dit['ID'], dit['fetch_time'] = zip(*tem)

writer = pd.ExcelWriter(file_path)
df = pd.DataFrame(dit)
# columns参数用于指定生成的excel中列的顺序
df.to_excel(writer, columns=['ID', 'machine_ID', 'put_time', 'fetch_time'], index=False, encoding='utf-8',
sheet_name='Sheet')
writer.save()

# save bug
file_path = './output_bug.xlsx'
writer = pd.ExcelWriter(file_path)
dit = {
'index': [],
'machine_index': [],
'start_time': [],
'end_time': [],
}
```

```
for item in self.dt:
dit['index'].append(item)
dit['machine_index'].append(self.dt[item][0])
dit['start_time'].append(self.dt[item][1])
dit['end_time'].append(self.dt[item][2])

df = pd.DataFrame(dit)
# columns参数用于指定生成的excel中列的顺序
df.to_excel(writer, columns=['index', 'machine_index', 'start_time', 'end_time'], index=False,
encoding='utf-8',
sheet_name='Sheet')
writer.save()




else:
dit = {
'ID1': [],
'first_ID': [],
'first_put_time': [],
'first_fetch_time': [],
'ID2': [],
'second_ID': [],
'second_put_time': [],
'second_fetch_time': []
}
for item in self.break_item:
dit['ID2'].append(item)
dit['second_ID'].append(-1)
dit['second_put_time'].append(-1)
dit['second_fetch_time'].append(-1)

for item in self.CNC_list:
item.produce_fetch_time()
if item.type == 0:
dit['ID1'] += item.list_index
dit['first_ID'] += [item.ID for _ in range(len(item.list_fetch_time))]
dit['first_put_time'] += item.list_put_time
dit['first_fetch_time'] += item.list_fetch_time
elif item.type == 1:
dit['ID2'] += item.list_index
dit['second_ID'] += [item.ID for _ in range(len(item.list_fetch_time))]
dit['second_put_time'] += item.list_put_time
dit['second_fetch_time'] += item.list_fetch_time
# sort
tem = zip(dit['ID1'], dit['first_fetch_time'], dit['first_ID'], dit['first_put_time'])
```

```
tem = sorted(tem)
tem = tem[:len(dit['second_ID'])]
dit['ID1'], dit['first_fetch_time'], dit['first_ID'], dit['first_put_time'] = zip(*tem)

tem = zip(dit['ID2'], dit['second_put_time'], dit['second_ID'], dit['second_fetch_time'])
tem = sorted(tem)
dit['ID2'], dit['second_put_time'], dit['second_ID'], dit['second_fetch_time'] = zip(*tem)

writer = pd.ExcelWriter(file_path)
df = pd.DataFrame(dit)
# columns参数用于指定生成的excel中列的顺序
df.to_excel(writer, columns=['ID1', 'first_ID', 'first_put_time', 'first_fetch_time',
'ID2', 'second_ID', 'second_put_time', 'second_fetch_time'], index=False,
encoding='utf-8',
sheet_name='Sheet')
writer.save()

# save bug
file_path = './output_bug.xlsx'
writer = pd.ExcelWriter(file_path)
dit = {
'index': [],
'machine_index': [],
'start_time': [],
'end_time': [],
}
for item in self.dt:
dit['index'].append(item)
dit['machine_index'].append(self.dt[item][0])
dit['start_time'].append(self.dt[item][1])
dit['end_time'].append(self.dt[item][2])

df = pd.DataFrame(dit)
# columns参数用于指定生成的excel中列的顺序
df.to_excel(writer, columns=['index', 'machine_index', 'start_time', 'end_time'], index=False,
encoding='utf-8',
sheet_name='Sheet')
writer.save()
```

```
class RGV(object):
def __init__(self, move_time=(0, 0, 0), do_cost_time=(0, 0), wash_time=0):
self.move_time = move_time
self.do_cost_time = do_cost_time
self.wash_time = wash_time
self.loc_pos = 0
self.hold_first = False
```

```
self.hold_index = -2

def is_holding(self):
return self.hold_first

def hold(self):
self.hold_first = True

def put(self):
self.hold_first = False

def get_now_pos(self):
return self.loc_pos

def get_go_pos_cost(self, pos):
distance = abs(self.loc_pos - pos)
if distance == 0:
return 0
else:
return self.move_time[distance - 1]

def do_go_pos(self, pos):
cost_time = self.get_go_pos_cost(pos)
# print(f'go to {pos}, where is machine:{pos*2,pos*2+1}, cost times:{cost_time}')
self.loc_pos = pos
return cost_time

def do_upload(self, ID):
if ID.ID // 2 == self.loc_pos:
# print(f'on pos:{ID.ID // 2},upload ID:{ID.ID} cost time {self.do_cost_time[ID.ID%2]}')

return self.do_cost_time[ID.ID % 2]
else:
raise Exception(f'error in upload, ID//2!=self.loc_pos,ID={ID.ID},loc_pos={self.loc_pos}')

def do_wash(self):
# print(f'wash and finish a process,cost time:{self.wash_time}')
return self.wash_time


if __name__ == '__main__':
pass
```