

基于粒子群和自适应遗传算法的RGV动态调度研究

2018年9月16日

摘要

本篇论文针对单、双工序的RGV智能加工系统，分别建立了粒子群算法模型、自适应遗传算法模型。针对故障调度问题，利用韦布尔分布进行分析。

对单工序的RGV动态调度问题，为达到单位时间产量最大的优化目标，建立粒子群算法模型，依据PSO算法对模型求解。最后得出三组单位时间最大产量分别为384个、370个、393个。在考虑故障的情况下，在原算法基础上进行智能调度，减少故障干扰。

对双工序的RGV动态调度问题，利用自适应遗传算法，定向筛选最优路径遗传个体，将其代入模型求解。最终得出三组单位时间最大产量分别为229个、212个、214个。在考虑故障调度的情况下，由统计分析得出第二道工序CNC故障数与产量成负相关。

随后进行静态调度检验，对最高效率路径进行探究与证明，从而检验动态调度代码实施的合理性与准确度，并对影响生产效率因素进行探讨。

最后进行对问题的延拓与优化。利用改进程序验证优化方案，试图还原真实生产过程并推广至更加普通的加工系统中。

关键词：自适应遗传算法，粒子群算法优化，动态多目标调度，智能调度

1 问题的提出

RGV智能加工系统由八台CNC计算机数控机床和1台RGV自动引导车组成。CNC负责原料的加工，RGV负责物料的呈送与清洗。在CNC端，RGV可以做到交换生熟料的作用，在上料传送带，RGV将CNC上的熟料或半成品拾入机械臂，用另一支机械臂将传送带上的生料装载到CNC中，完成交换。在下料传送带，RGV将CNC上的熟料拾入机械臂，另一支机械臂将清洗后成料放入传送带或将半成品装载到CNC上，完成交换。CNC有三种状态，工作、等待和上料。RGV有四种状态，移动、等待、上料、清洗。题目给出三种情况：

- (1) 单工序加工。即所有CNC都能将生料加工为熟料。
- (2) 双工序加工。该情况下八台CNC被分成两类，一部分加工第一道工序，将生料加工为半成品，另一部分加工第二道工序，将半成品加工为熟料。
- (3) 该情况是对前两种情况的延伸，在每次加工过程中有百分之一的概率发生故障，故障发生后物料报废，10-20分钟后恢复正常。

总体来说CNC和RGV的协作问题，为了使物料生产效率最高。需要对RGV进行合理的控制与规划。本题针对RGV智能系统为我们设计了两个问题：

- (1) 为RGV建立一个动态模型和算法，给出一般问题下的RGV行动路径和工作流程。
- (2) 利用表格中的三组数据，将模型算法应用与三种情况中，得出具体的物料追踪数据，检验模型和算法的实用性与有效性。

另外，下面我们会将“单工序”、“双工序”、“含故障”称为“问题1”、“问题2”、“问题3”。

2 问题的假设

在刀具不同的情况下，我们想利用遗传算法，进行对最优解的探究。在此之前我们首先做出两点假设：

- (1) RGV的运作顺序是第一道工序和第二道工序的交叉排列，我们认为系统中RGV不会连续去同一道工序工作，因为其会导致RGV只有一条空余手臂而不足以完成上下料操作。
- (2) 装配线不允许发生无料缺货，即CNC只有两种状态：工作、工作完成等待RGV、上料。

3 问题的分析

3.1 问题1

对于问题一，RGV小车可以执行4种动作，包括移动，等待，上下料和清洗。通过对小车执行这四个动作的时间进行累加，我们可以计算在一个班次中得到RGV的效率。

相比于最短路径的算法（TSP），我们采用的粒子群算法建模拥有更快的运行效率和收敛的速度。

对于故障调度方面，我们利用韦布尔分布来描述机器发生故障的情况，用均匀分布来判定每次故障维修所需要的时间。然后，利用粒子群算法在较长的一段时间内决定最优的调度来应对故障的情况。

3.2 问题2

精简版：在该假设下，RGV在工作一定时间后将陷入循环，我们将该循环的行走路径作为个体，建立一个含有N个具有相同循环长度的个体的种群。接着我们建立适应度函数计算每个个体的时间，利用轮盘赌方法赋予低耗时个体更高的权重，从而建立子代种群。再通过交叉、变异操作，对种群进行变异。由此反复，便可得出局部最优解。

由于更快的RGV路径个体有更高的概率存活到下一代，所以种群朝着平均效率更快的方向演进。

3.3 问题3

4 建模

4.1 模型建立

在上述假设下，当单周期时间最短时，即为RGV系统调度问题最优方案。可描述为： $\min \sum_{i=1}^n T_{task}^i$ 。其中 T_{task}^i 表示第*i*个任务所需时间，该时间可表示为 $T_{task} = T_{move} + T_{wait} + T_{load} + T_{clean}$ 。其中 T_{move} 代表RGV从前位点移动到现位点的时间， T_{wait} 代表RGV在CNC前等待的时间， T_{load} 代表RGV在CNC前上下料所用时间， T_{clean} 代表清洗时间。

这里我们定义RGV与CNC个各类操作及其所需时间：

系统作业参数	时间规定	操作数
RGV移动1个单位所需时间	t_{move1}	a
RGV移动2个单位所需时间	t_{move2}	b
RGV移动3个单位所需时间	t_{move3}	c
CNC加工完成一个一道工序的物料所需时间	t_{finish}	d
CNC加工完成一个两道工序物料的第一道工序所需时间	t_{first}	e
CNC加工完成一个两道工序物料的第二道工序所需时间	t_{second}	f
RGV为CNC1#, 3#, 5#, 7#一次上下料所需时间	t_{load1}	g
RGV为CNC2#, 4#, 6#, 8#一次上下料所需时间	t_{load2}	h
RGV完成一个物料的清洗作业所需时间	t_{clean}	k

表 1: 操作所需时间

RGV的调度算法主要工作即，当某一CNC完成当前工作即加入当前等待队列，调度算法根据最少时间原则，分配RGV小车为等待的CNC服务。我们规定轨道上运行的RGV小车*N*辆，记 N_i 为第*i*辆小车

4.2 目标函数

找到一种路径，使 $\sum_{i=1}^n T_{task}^i$ 最小。

4.3 约束条件

智能RGV加工系统对该模型存在以下约束条件：

(1) 最小循环时间 $\min \sum_{i=1}^n T_{task}^i$ 必须大于每一个CNC的加工时间。

5 单工序动态调度

5.1 算法与模型

首先，将一个班次8小时划分为几个周期，在每个周期内通过利用粒子群算法的思想，得到周期内效率最高的方案，并将这一周期结束后CNC的状态记录（包括是否工作或者空闲，还有多少时间完成工作，是否故障，还有多少时间能够维修结束等等），传递给下一个周期，直到时间达到8小时。

5.2 数据表示

对于单工序情况，我们利用粒子群算法进行建模，得到高效率的调度。

首先，我们对于每一个粒子，我们产生一个 $1 \times n$ 的向量来表示RGV小车在一个周期内的移动路径。每一行从左到右分别表示RGV先后移动经过的CNC顺序。一共有 m 个粒子，这样整个群体可以生成一个 $m \times n$ 的矩阵。

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix}$$

对于每次迭代，我们记录每一个粒子所经历过的局部最优矩阵 $pbest(n \times m)$ ，和全局最优向量 $gbest$ 。

5.3 PSO的应用

由于PSO问题原本是应用于连续性问题的，但是单工序的动态调度是离散的。原本的公式：

$$v_{id}^k = w v_{id}^{k-1} + c_1 r_1 (pbest_{id} - x_{id}^{k-1}) + c_2 r_2 (gbest_{id} - x_{id}^{k-1})$$

$$x_{id}^k = x_{id}^{k-1} + v_{id}^{k-1}$$

已经不再适用。

(其中， v_{id}^k 表示第 k 次迭代粒子 i 飞行速度矢量的第 d 维分量。

x_{id}^k 表示第 k 次迭代粒子 i 位置矢量的第 d 维分量。

c_1, c_2 表示加速度常数。

r_1, r_2 是两个随机数，取值范围 $[0, 1]$ 。

w 表示惯性权重，非负数。)

所以我们将PSO的方法做了一定的改变来应用于本问题。

首先，我们定义了一些新的算子来适用这个方法。我们定义 \ominus ，一种类似于原来 $pbest_{id} - x_{id}^{k-1}$ 的操作来描述两个移动路径的差，得到的结果是一组数对。对于每个数对 (a, b) ， a 表示 $pbest_{id}$ 与 x_{id}^{k-1} 不同的位置， b 表示该位置上 $pbest_{id}$ 与 x_{id}^{k-1} 的位置之差。例如：

$$A = [1, 2, 3, 4, 5, 6, 7, 8]$$

$$B = [1, 2, 1, 4, 5, 6, 7, 6]$$

$$A \ominus B = [(2, 2), (7, 2)]$$

其中，位置的计数从0开始。数对组中数对的顺序不分前后。

对于原来公式中的+，改变后的PSO有两种含义：

1. 如果，一个实数乘以一个数对组，其中，那个实数代表概率。例如p乘以 $A \ominus B$ 。我们将这个运算定义为一个新的算子 \otimes 。它的作用是对于数对组中的每一个数对，以p的概率选取(其中p介于0到1之间)，得到的结果仍然是一个数对。例如，

$$0.5 \otimes [(1, 2), (3, 4)] \text{ 得到的结果就有可能} [(1, 2)]$$

2. 如果，那个实数代表的是一个系数。例如c乘以 $A \ominus B$ 。我们将这个运算定义为一个新的算子*。它的作用是对于数对组中的每一个数对的第二项b的值乘以c，然后向下取整，得到的结果仍然是一个数对。一般c的值是介于0到1之间。例如，

$$0.5 * [(1, 2), (3, 4)] = [(1, 1), (3, 2)]$$

对于原来公式中的+，改变后的PSO也有两种含义：

1. 如果操作符的两边都是数对组的话，那么+就相当于集合运算中取并集的运算 \cup 。例如：

$$[(1, 2)] + [(3, 4), (5, 6)] = [(1, 2), (3, 4), (5, 6)]$$

2. 如果等号的两边，一边是移动的路径，另一边是数对组的话。我们将这种运算定义为 \oplus 。它将数对组所描述的改变应用到移动路径上。对于，数对组中的每一个数对(a,b)，移动路径上a的位置的值就要加上b。例如：

$$A = [1, 2, 3, 4, 5, 6, 7, 8]$$

$$B = [(1, 2), (3, 4)]$$

$$A \oplus B = [1, 4, 3, 8, 5, 6, 7, 8]$$

最后，原来公式中速度就可以表达为数组队，描述两个移动路径之间的差异。

原来的PSO公式就可以写为：

$$v_{id}^k = w \otimes v_{id}^{k-1} + c_1 * r_1(pbest_{id} \ominus x_{id}^{k-1}) + c_2 * r_2(gbest_{id} \ominus x_{id}^{k-1})$$

$$x_{id}^k = x_{id}^{k-1} \oplus v_{id}^{k-1}$$

其中， c_1, c_2 是系数， r_1, r_2 是概率。

5.4 粒子的初始化

粒子位置的初始化：我们采用泊松分布的方法来初始化矩阵中的每个元素，范围从1-8。

粒子初始速度的初始化：首先，我们用均匀分布得到一个1到8之间的随机数，来决定初始速度中数对组的个数。然后，对数对组中每一个数对通过随机产生。

5.5 粒子适应值的计算

针对每一个粒子的移动路径，通过计算这个周期内 $\min \sum_{i=1}^n T_{task}^i$ 。其中 T_{task}^i 表示第*i*个任务所需时间，该时间可表示为 $T_{task} = T_{move} + T_{wait} + T_{load} + T_{clean}$ 。其中 T_{move} 代表RGV从前位点移动到现位点的时间， T_{wait} 代表RGV在CNC前等待的时间， T_{load} 代表RGV在CNC前上下料所用时间， T_{clean} 代表清洗时间。通过比较哪一个粒子所用的时间最少，来决定哪个粒子的效率最高，从而决定是否更新个体最优和群体最优。

5.6 收敛情况

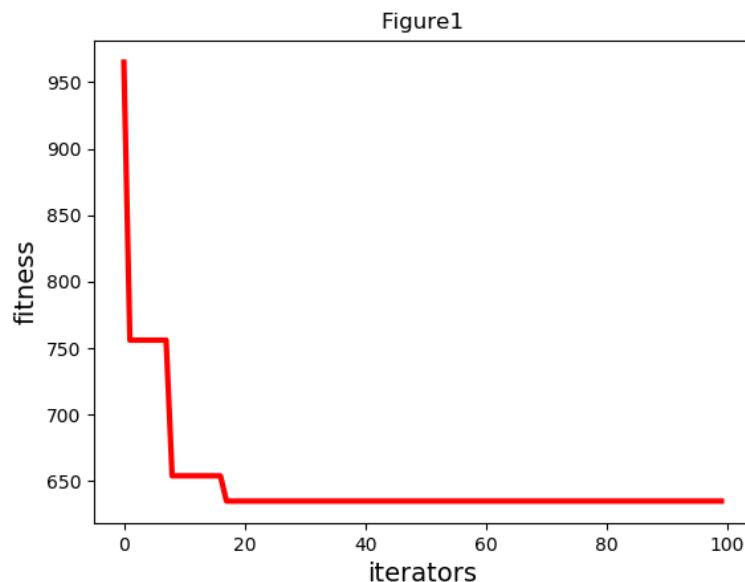


图 1: PSO单周期的收敛趋势

通过上图可知，

5.7 算法步骤

1. 判断是否达到班次的最大时间，如果是，结束；否则开始2
2. 周期开始，导入CNC的状态
3. 初始化粒子群
4. 计算每个粒子新位置的适应值
5. 根据新的适应值得到个体最优和全局最优

6. 根据改进过的公式:

$$v_{id}^k = w \otimes v_{id}^{k-1} + c_1 * r_1(pbest_{id} \ominus x_{id}^{k-1}) + c_2 * r_2(gbest_{id} \ominus x_{id}^{k-1})$$

$$x_{id}^k = x_{id}^{k-1} \oplus v_{id}^{k-1}$$

来更新速度和位置

7. 判断是否达到最大的迭代次数。如果没有，返回4，继续迭代。否则记录当前时间，与CNC状态，进入下一个周期,返回1。

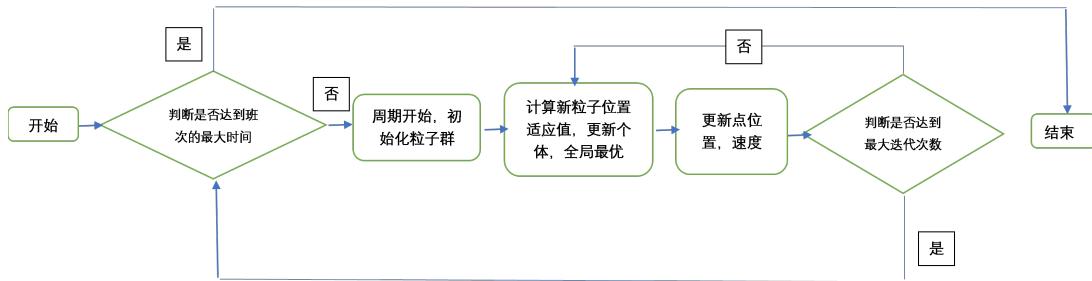


图 2: 流程图示意

5.8 算法优化

针对粒子群算法有时会收敛过快，而收敛在局部最优的情况。我们还除了正常的粒子外，还加入了一些随机的粒子，来减轻这种情况。随机粒子的数目与正常粒子数目，可以自适应的变化。

5.9 故障分析

组数	CNC故障数量	完成件数
第一组	10	360
第一组	11	372
第一组	7	365
第二组	15	353
第二组	11	351
第二组	5	355
第三组	4	385
第三组	10	374
第三组	15	372

表 2: 单工序故障模拟

通过与未发生故障的作业量比较，第一组384个，第二组370个，第三组393个。可以发现：

1. 以1%的概率发生故障时，该班次减少生产量在0-20个左右。
2. 经过统计，故障数与生产量存在着一定的负相关和线性关系。

6 模型的优缺点

6.1 模型的优点

1. 基于粒子流算法的单工序调度模型，拥有较快的收敛性，能更快的得到较优的结果。

6.2 模型的缺点

1. 粒子流算法可能会有过快收敛的问题，有存在错过最优解的可能。但是，我们同过加入随机粒子和自适应权重等方法来减轻这个可能。

7 双工序的动态调度算法

两道工序的情况相对比较复杂所以这里我们先做一个简单分析。但其中第一组数据的情况相对比较简单，因为第一组中两道工序的工作时间近乎相等，所以还是比较对称的情况。我们选择偶数CNC做第一道工序，选择奇数CNC做第二道工序，采用的依旧是类似图1的路径，只不过因为CNC2是第一道工序，这次的出发点是CNC2。相比于单工序问题，每个循环只有第二道工序的4个CNC有清洗环节，所以我们只需将原公式的 $t_{clean} \times 8$ 修正为 $t_{clean} \times 4$ 。计算所得的两个值为411和414，与工作时间400和378比较。取出最大值414。加上一次工作时间28s，最终最优循环为442秒。与遗传算法吻合得刚刚好。

第二组和第三组的情况更加复杂，因为这两种情况中两道工序的工作时间差异很大，如果给两道工序均分配4个CNC，速度较快的工序将会经常处于闲置状态。所以我们应该使八台CNC尽可能的同时工作，提高整体效率。

RGV的调度算法主要工作即，当某一CNC完成当前工作即加入当前等待队列，调度算法根据先前确定的优化目标，分配RGV小车为等待的CNC服务。对于每一个生料我们规定了相应的生产集合，集合主要由我们上文确定的操作数所构成，各道工序会占用相应的CNC和RGV资源，并且由于工序的顺序性双工序的动态调度就要比单工序的动态调度复杂很多。所以RGV调度算法的最终目标即为合理安排CNC, RGV资源和加工时间，达到、textbf{单位班次成料生产最大化}和单位班次RGV路程最短的优化目标。由于第一道工序和第二道工序所需时间不同我们还需对第一道和第二道工序分别占有的CNC数量和位置进行优化。

系统作业参数	时间规定	单位
RGV单位班次操作所需时间	T_{opt}	s
RGV单位班次生成的成料个数	N	per
RGV单位班次移动的总路程	l_{move}	m
CNC加工完成一个两道工序物料的第一道工序所需时间	t_{first}	s
CNC加工完成一个两道工序物料的第二道工序所需时间	t_{second}	s
CNC工作台的工序列表	$M = \{i \in \{0, 1\}\}$	无

表 3: 参数约定

这里 M 长度为8，假设 $M = \{0, 1, 0, 1, 0, 1, 0, 1\}$ 则代表CNC#1可以加工第一道工序。所以我们的优化目标为：

$$\min \alpha T_{opt} + \beta l_{move} + \gamma/N = \sum_1^m (t_{move} + t_{clean} + t_{wait} + t_{load}) + \sum_{i=1}^m l_{move}^i + \gamma/N$$

subject to : $N = 1, 2, 3, 4\dots, \alpha > 0, \beta > 0, \gamma > 0, T_{opt} > t_{first} + t_{second}$

7.1 CNC 任务分配

由于第一道工序和第二道工序的时间不同我们在优化总体时间和生产数量时先对CNC任务进行分配。首先进行数量分配，令 $number_{first}$ 为第一道工序的CNC 数量， $number_{second}$ 为第二道工序的CNC 数量。则规定：

$$\frac{number_first}{number_second} = \frac{t_{first}}{t_{second}}$$

所以根据题干里面的三组数据这里给出最有的CNC人物分配方案。



图 3: CNC最优数量与位置分配

7.2 多目标自适应遗传算法优化

对于非凸的问题我们很难在有限的时间内求出原问题的最优解所以我们这里使用启发式遗传算法来进行原问题的求解。受到自然界的启发人们对生物的各种生存特性和激励进行研究和模拟，遗传算法通过初始化种群运用特定的方法进行种群的选取不断进行变异和交叉从而使最终的解集趋于全局最优解。

由于传统的遗传算法的交叉和变异概率大部分依靠经验取值，想要获得很好的性能需要大量调试和经验。但是由于调度算法需要实时性所以不能运用传统的遗传算法。本文使用的、textbf{自适应遗传算法}自动适应个体的变异和交叉率，既让表现良好的个体有更大的机会去繁殖后代也给现在表现差的个体留有机会，防止程序陷入局部最优解。当个体差异小时程序自动调整变异率使得群体的差异变大。

7.2.1 个体编码

与传统的遗传算法不同的是由于这里需要考虑工序的顺序和工序的先后完成顺序，我们不能简单的采用二进制的编码方式，这里我们采用的二维矩阵符号编码的方法。由于我们观察到RGV的运作顺序是第一道工序和第二道工序的交叉排列，我们认为系统中RGV不会连续去同一道工序工作，因为其会导致RGV只有一条空余手臂而不足以完成上下料操作。我们定义一个个体P，其型式为 $2 \times n$ 的矩阵，其含义为一个周期内RGV的移动路径。每一列的上下两部分分别代表RGV先后经过的两个CNC。RGV从左到右一列一列执行矩阵，一个周期共经过 $2n$ 个CNC。其次，我们定义一个种群Q， Q是由N个个体P构成的。

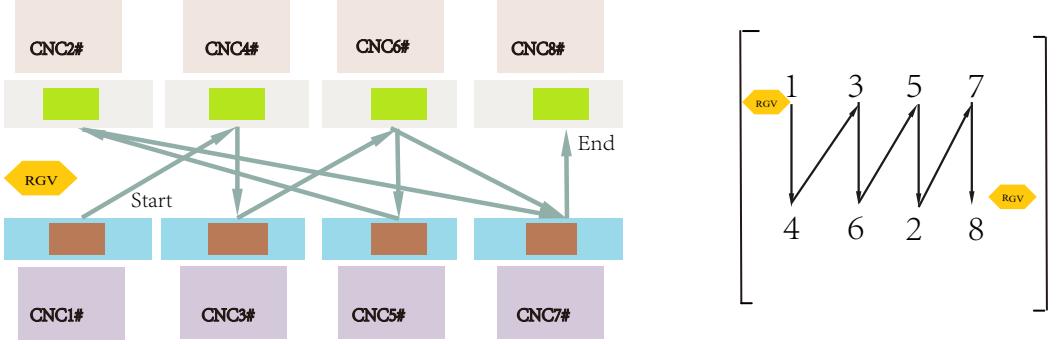


图 4: 个体编码举例

7.2.2 适用度评价函数

我们运用适用度函数来评价每个个体的好坏，在本题中我们主要优化我们上文提到的目标函数即路程最短，生成成品数量最多。这里我们假设：

$$f_{time} = \min \sum_{j=1}^m (t_{move} + t_{clean} + t_{wait} + t_{load})$$

$$f_{num} = \max \sum_{i=1}^8 N_i$$

统一函数一和函数二的区间我们将其映射到同一区间内：

$$f_1(x) = \frac{f_{time}(x) - f_{time\ min}(x)}{f_{time\ max}(x) - f_{time\ min}(x)} f_2(x) = \frac{f_{num}(x) - f_{num\ min}(x)}{f_{num\ max}(x) - f_{num\ min}(x)} \quad (1)$$

我们假设种群数量为N则自适应权重为：

$$\alpha = \frac{1}{f_{1\ max}(x) - f_{1\ min}(x)} \beta = \frac{1}{f_{2\ max}(x) - f_{2\ min}(x)} \quad (2)$$

$$\alpha = \frac{\alpha}{\alpha + \lambda \times \beta} \quad \beta = \frac{\lambda \times \beta}{\alpha + \lambda \times \beta} \quad (3)$$

这里我们假设 $\lambda = 0.1$

7.2.3 交叉操作

由于遗传算法容易陷入局部最优，当种群中多数的个体达到局部最优解时种群很难再达到全局最优解，为了能够跳出全局最优解我们引入了交叉操作从而使得一部分个体可以继承父代的优点但是经过变异可能达到全局最优。我们通过上文的介绍引入自适应的交叉变异率，当种群中绝大部分个体表现良好时我们增大交叉率，当种群中大部分表现不好时我们自动减小交叉率。交叉率的计算如下面所示：

$$p_{cross} = \begin{cases} m_1 \times \sin(\frac{\pi}{2} \times \frac{f_{max} - b}{f_{max} - f_{avg}}) & b > f_{avg} \\ m_2 & b < f_{avg} \end{cases}$$

这里 b 为我们的个体适应度， f_{avg} 为个体的平均适应度。

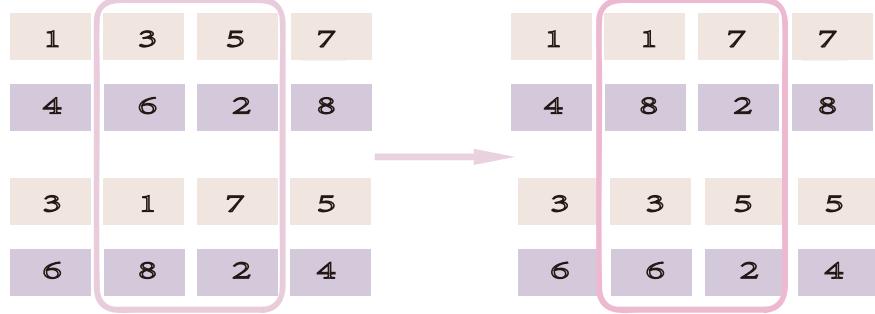


图 5: 交叉操作举例

7.2.4 变异操作

变异操作和交叉操作类似也可以通过变异操作来减少遗传算法卡在局部最优解的概率减少。变异操作的流程如下随机选取1,2,3,4中的数字,然后对当列的数据进行变异。第一行由于是第一道工序的所以选取(1,3,5,7)中的数据, 第二行是第二道工序的所以选取(2,4,6,8)中的数据。再程序的运行过程中我们根据种群的表现来动态的进行变异率的调整当种群中绝大部分的个体表现良好且趋于相同时我们将种群的变异率调大反之亦然。下面给出了程序运行过程中变异率的动态调整公式:

$$p_{mutation} = \begin{cases} m_3 \times \sin\left(\frac{\pi}{2} \times \frac{f_{max}-b}{f_{max}-f_{avg}}\right) & b > f_{avg} \\ m_4 & b < f_{avg} \end{cases}$$

这里 b 为我们的个体适应度, f_{avg} 为个体的平均适应度。



图 6: 变异操作

7.2.5 变异和交叉个体的选取

由于种群需要进化所以我们需要选择出个体适应度最好的个体来进行下一次的迭代过程。在本程序中我们采用了轮盘赌的算法来选择个体适应度最强的个体来进行交叉的操作, 而变异的操作则任意一个个体都可以进行。轮盘赌的具体操作如下:

(1) 计算每一个个体的个体适应度 $f(i = 1, 2, 3..M)$ 其中 M 为种群的大小

(2) 然后计算每个个体被遗传到下一代的概率这里计算方式如下:

$$p(x_i) = \frac{f(x_i)}{\sum_{j=1}^M f(x_j)}$$

(3) 在[0,1]中随机生成一个概率

(4) 计算个体的累计概率:

$$q_i = \sum_{j=1}^i p(x_i)$$

(4) 比较 q_i 与生成的随机数进行比较，然后进行相应的选择

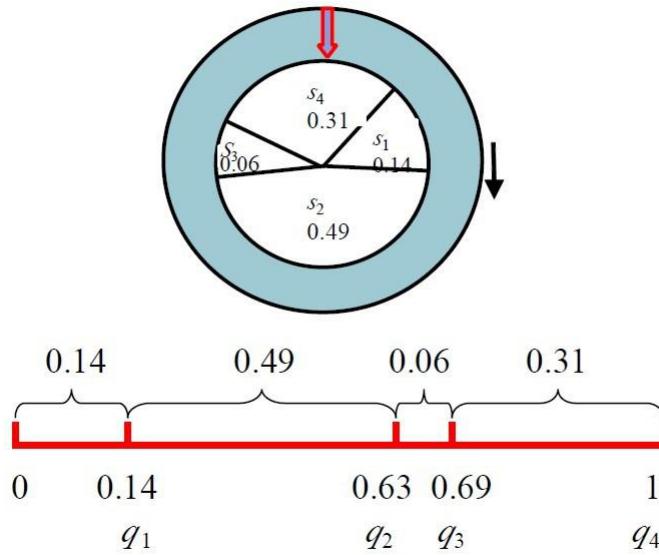


图 7: 轮盘赌选择操作

7.2.6 双工序算法流程

对于问题二，我们将用遗传算法建模，该算法可以向高效路径逼近，适合该问题。

首先，我们定义一个个体P，其型式为 $2 \times n$ 的矩阵，其含义为一个周期内RGV的移动路径。每一列的上下两部分分别代表RGV先后经过的两个CNC。RGV从左到右一列一列执行矩阵，一个周期共经过 $2n$ 个CNC。其次，我们定义一个种群Q，Q是由N个个体P构成的。

接下来便是如何实施我们的遗传算法：

- (1) 随机出N个个体P，组成一个种群Q。
- (2) 建立一个适应度函数 $f(P)$ ，输出的结果为RGV移动一个周期的时间。
- (3) 在种群Q中，利用轮盘赌方法，即赋予更快的个体P更高的权重，依靠该权重一个一个重新筛选出新的N个个体P，组成新的种群Q。
- (4) 在新种群Q中，每个个体与另一个体交换一次矩阵中的一列，随后每个个体有一定概率变换矩阵中的一列。我们称种群Q中的个体发生了突变，随后便产生突变种群Q。
- (5) 重复(2)-(4)n次。

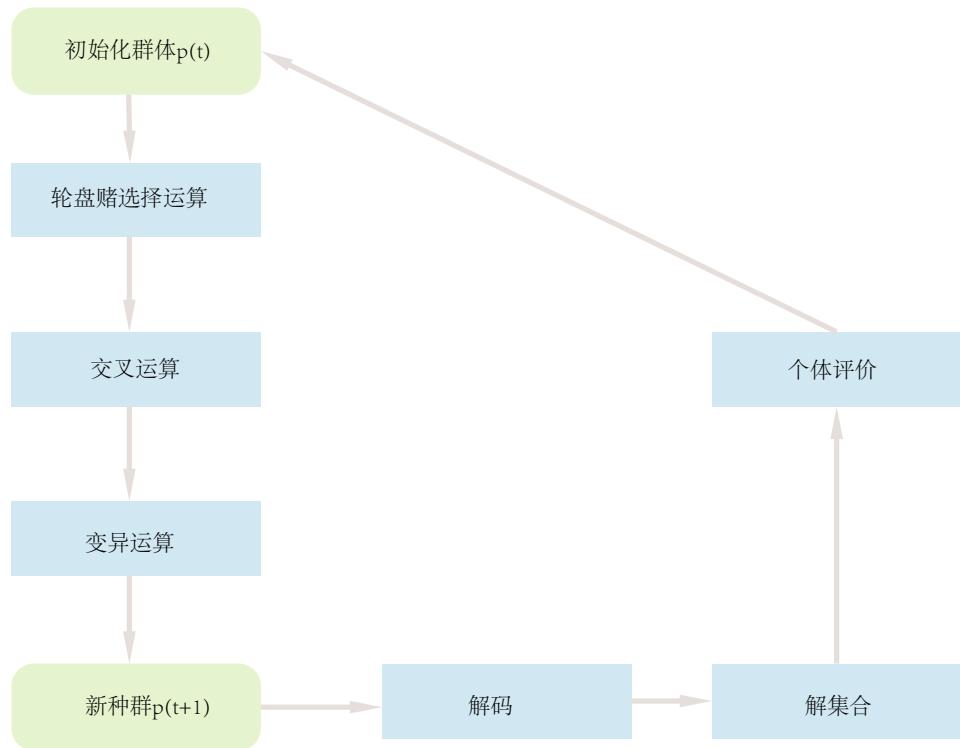


图 8: 算法流程图

7.2.7 双工序动态遗传算法结果

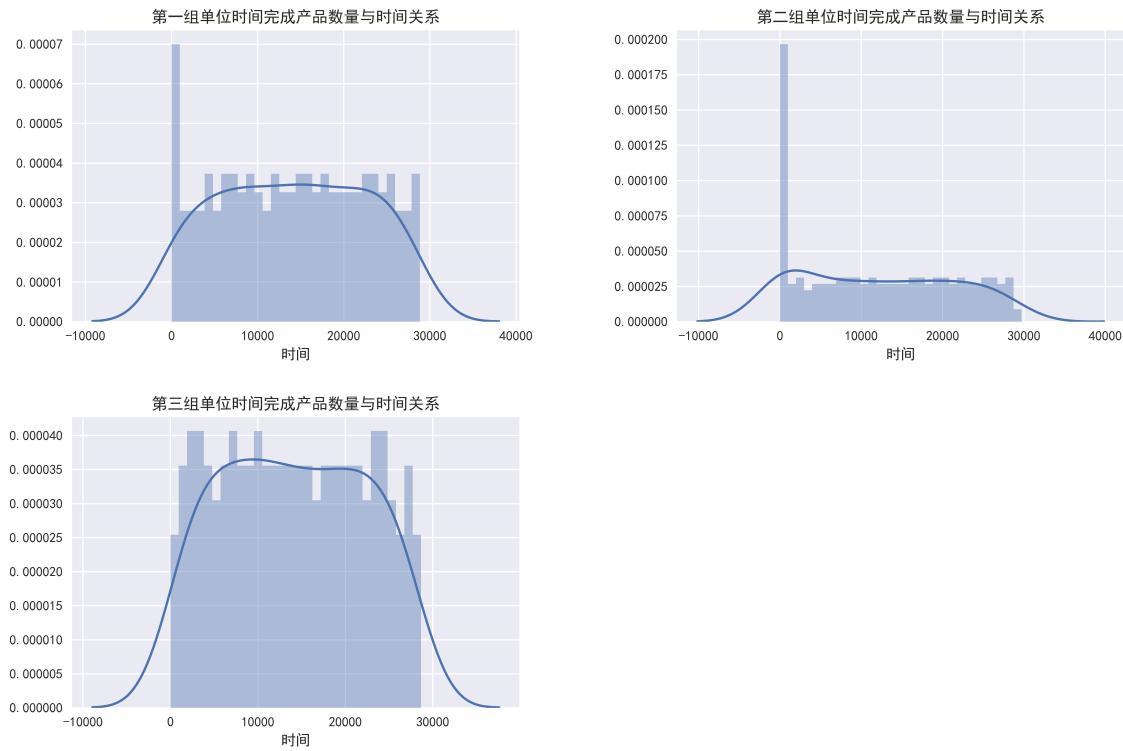
这里我们列出当没有错误发生时，自适应遗传调度算法的运算结果，我们通过python进行最终结果的模拟，运用表格中的三组数据当作程序中的初始值进行最终的求解，求解的结果如下：

组号	产量	排列方式
1	229	
2	212	
3	214	

图 9: 遗传算法结果

算法的出事种群数量为200个，初始变异率0.4，初始交叉率0.8由于本程序使用了自适应的遗传算法所以可以在程序的运行过程中自动的收集种群的信息进而对变异率和交叉率进行自适应的改变，使得算法的性能有了很大的提升。我们每次迭代进化5000次，为了达到更好的效果可以适当的增加种群的迭代次数。本题的CNC数量和RGV数量都不多可以用穷举暴搜进行模型求解，但是当CNC数量变大后模型的优势就可已显现出来。

我们继续对算法的性能进行分析，最好的算法必然是CNC尽可能进行工作而，RGV有求必应。我们列举随着时间的增加单位时间算法可以生成的成品数量。我们分别列举了第一组和第二组，第三组的单位时间成品数量和时间的关系：



从上面三幅图中我们发现在第一个小周期的时候算法的效率最高这是由于我们初始化的时候自动为每一个及其装好了一个成品，在初始化过后模型趋于稳定可以发现每个时间段内模型完成的数量大体形同，证明我们的堵塞状况出现的较少，CNC一直处于工作状态。这时程序的运行效率最高。

7.2.8 双工序动态调度的故障分析

由于题目中每台机器的故障率为0.01，经过一个班次的运行后我们等到统计平均值为大概会发生5次故障，这里我们按照第一组中的数据，运行了6次算法后系统可以最终完成成料的数量和总故障数如下表所示：

总故障	第一道工序故障	第二道工序故障	完成件数
6	3	3	224
4	1	3	226
3	1	2	230
5	2	3	225
9	4	5	219
6	3	3	226

表 4: 双工序故障模拟

当没有故障时我们的CNC在八小时内大概可以做出230-240间成品在发生故障时我们的算法仍然可以达到220以上的成品率课件虽然在每个小循环上遗传算法可能没有找到当前的最优解，但是从整体上看来我们的算法还是可以达到不错的性能。通过观察最终的完成度我们发现，当发生故障且故障率在0.01时算法最终可以完成的数量要比无故障时小10个左右。从数据中可以看出故障数和最终完成的总作业量呈现负相关的关系，而二号工作台的故障数和总完成数的负相关性更强。

7.2.9 双工序的仿真结果

通过上文的讨论我们知道如果第一道工序和第二道工序所用时间相差较大则CNC的分配的最优解不再是4个第一道工序和4个第二道工序。在这里我们运用我们的遗传算法对第二三组数据进行模拟仿真，当我们分配给第一道工序和第二道工序的CNC数量变化时，在八小时内得到的最终成品数量的变化。和我们上文提出的结论进行匹配。我们进行验证的数据如下：



图 10: 假设的CNC最优数量与位置分配

- 第二组：1 : $N_{first} = 3, N_{second} = 5$, 2 : $N_{first} = 2, N_{second} = 6$
- 第三组：1 : $N_{first} = 5, N_{second} = 3$, 2 : $N_{first} = 6, N_{second} = 2$

我们分别以第二组和第三组数据为依据运行我们自行编写的模拟程序，下面是程序的运行结果：

组数	第一道工序CNC数量	第二道工序CNC数量	完成件数
第二组	4	4	229
第二组	2	6	124
第二组	3	5	220
第三组	4	4	200
第三组	6	2	187
第三组	5	3	226

表 5: 双工序故障模拟

从上面的数据我们可以看出第二组分配到4个第一道工序和4个第二道工序为最优情况，模拟时最多加工了229个当然我们通过式子计算的3: 5的比例达到了220，这是由于第二组的两道工序时间差异还不够明显所以在模拟的时候4: 4，和3: 5的模型差异不大。但是反观第三组的数据当分配到5个第一道工序和3个第二道工序为最优情况，达到了226个显著的高于其它两种比例。经过这次模拟我们可以发现CNC的分布和数量对模型的显著影响。

8 静态调度检验

因为动态调度是靠python实行，有编译错误的风险，所以我们增加进行静态调度检验步骤。

因为CNC的工作时间远长于RGV的移动、上下料、清洗时间，所以在循环中，RGV花费时间去等待CNC是不可避免的，而CNC等待RGV则会造成时间浪费。所以静态调度的核心就是尽可能使全部CNC都在工作，且使RGV等待CNC的时间最小。

8.1 问题1

对一道工序的情况进行动态调度相对容易，因为所有CNC是几乎对称的。最优路径是极易构想的，如下图。首先从CNC2上料结束开始计时，到第二次开始上料前截止，我们计算一个循环的时间为 $25 * 8 + 20 * 3 + 46 + 28 * 3 + 31 * 4 = 514$ 。这段时间小于CNC2的工作时间560，所以我们取循环时间560s。接下来我们从CNC1上料结束开始计时，到第二次开始上料前截止，我们计算一个循环的时间为 $25 * 8 + 20 * 3 + 46 + 28 * 4 + 31 * 3 = 511$ 。这段时间小于CNC1的工作时间560，所以我们取循环时间560s。其他奇数机器与CNC1相同，偶数机器与CNC2相同。但560并不是最终的循环时间，因为这个560是从第一轮的上料结束时刻计时，直到第二轮上料开始时刻。所以还应加上一次上料时间28s。因此，最后得出的最短循环时间一定是588s。因为这个过程中所有CNC都在全程工作，没有等待时间。

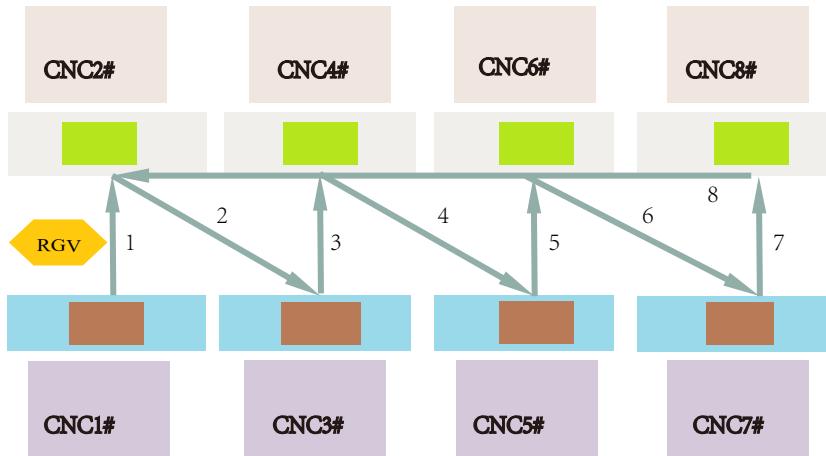


图 11: 最优路径

通过以上的计算，我们总结出了图1路径的通式解法：

$$t_{clean} \times 8 + t_{one-step} \times 3 + t_{three-step} + t_{odd} \times 3 + t_{even} \times 4$$

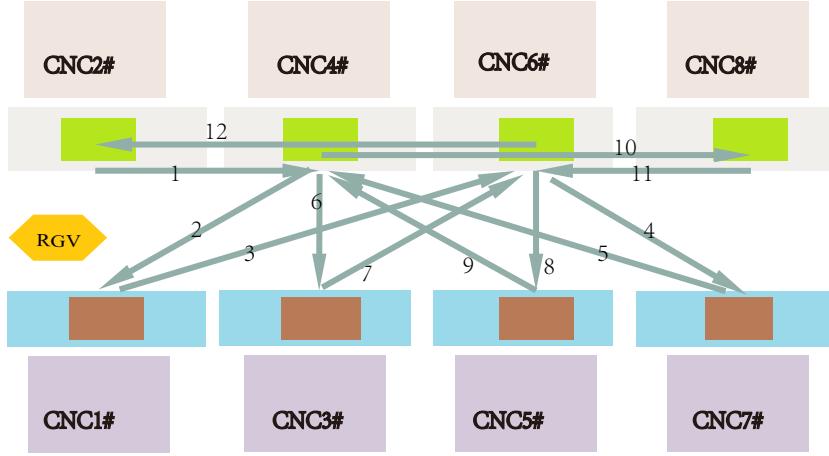


图 12: 第三组数据最优路径

$$t_{clean} \times 8 + t_{one-step} \times 3 + t_{three-step} + t_{odd} \times 4 + t_{even} \times 3$$

然后取他们与 $t_{CNC-working}$ 的最大值。对于第二组，上述两个值分别为598和593，与工作时间580比较，三者最大值为598。再加上一次工作时间30，最优循环为628秒。对于第三组，上述两个值分别为509和504，与工作时间545比较，三者最大值为545。再加上一次工作时间27，最优循环为571秒。

8.2 问题2

两道工序的情况相对比较复杂。但其中第一组数据的情况相对比较简单，因为第一组中两道工序的工作时间近乎相等，所以还是比较对称的情况。我们选择偶数CNC做第一道工序，选择奇数CNC做第二道工序，采用的依旧是类似图1的路径，只不过因为CNC2是第一道工序，这次的出发点是CNC2。相比于单工序问题，每个循环只有第二道工序的4个CNC有清洗环节，所以我们只需将原公式的 $t_{clean} \times 8$ 修正为 $t_{clean} \times 4$ 。计算所得的两个值为411和414，与工作时间400和378比较。取出最大值414。加上一次工作时间28s，最终最优循环为442秒。与遗传算法模拟的结果吻合得刚刚好。

第二组和第三组的情况更加复杂，因为这两种情况中两道工序的工作时间差异很大，如果给两道工序均分配4个CNC，速度较快的工序将会经常处于闲置状态。所以我们应该使八台CNC尽可能的同时工作，提高整体效率。为了达到两道工序总工作效率接近，自然应给较快的工序分配较少的机器，具体的分配方案应由两道工序的时间比决定。

对于第二组情况，两工序工作时间比为 $280 : 500 = 2.87 : 5.13$ 。因此我们选择5 : 3的配比，即3台第一道工序CNC，5台第二道工序CNC。为缩减RGV的总行程，我们将多出的一台CNC设在中间两个位置之一。在5 : 3的方案中，循环节的长度会变化，变为每生产15个物料为一周期。因此，通过静态调度，我们可以检验模拟结果中的工序分配与循环节长度是否正确，结果是吻合的。但我们无法手动计算出具体的最优循环时间，因为该循环过于复杂，只能通过程序计算。

第三组情况同理，两工序工作时间比为 $455 : 182 = 5.71 : 2.28$ 。因此我们选择2 : 6的配比，即6台第一道工序CNC，2台第二道工序CNC。循环长度为6个物料一周期。经过与模拟结果比对，也是吻合的。路径如下图所示。

9 模型的优化

在建模的过程中，我们发现，尽管RGV智能加工系统采取最优的工作方案，其在某些条件下的工作效率仍不尽如人意。因此，如果对该智能加工系统进行某些改进，如增减RGV的数量，改变轨道运作模式等。该模块就几种优化方案进行了讨论比较。

9.1 优化原则

从实际的工业生产角度出发，系统的优化需要较小的资本投入和较大的收益增量。所以，在优化时应注意投入资本的转化率。因此，系统的优化结果需要满足以下条件：

- (1) 投入资本转化效率增加，即产品增量与投入增量的比值应大于原产量与原投入量的比值。
- (2) 不能改变CNC和RGV的工作流程。

9.2 优化方案

9.2.1 增减RGV数量

在8台CNC的情况下，如果CNC的工作时间很长，RGV会经常处于闲置状态，此时增加RGV的数量对于系统提高生产效率的帮助甚微，投入的RGV成本没有收到应有的增益，不符合优化原则。而如果CNC的工作时间较短，RGV供不应求，增加RGV可能具有较优的优化效果。

首先，在多RGV情况下，我们限制RGV不能相撞，但可以双向移动。在此情况下，我们对原程序进行了修正。分别对两种情况进行了测试(均为单工序模式)：

- (1) CNC加工时间为800秒，其他条件与题目第一组数据相同。
- (2) CNC加工时间为100秒，其他条件与题目第一组数据相同。

接着，我们将每种情况对应的1、2、3、4个RGV的循环时间记录下来，最终结果用图表表示如下：

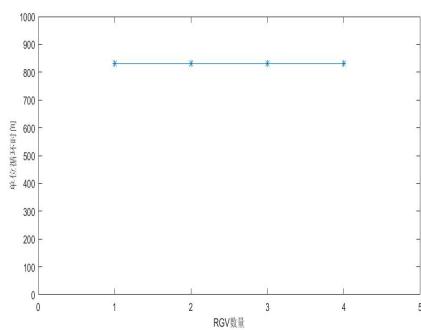


图 13: (a) CNC工作时间为800秒

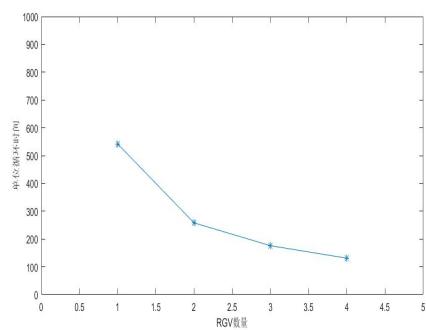


图 14: (b) CNC工作时间为100秒

图 15: 两种情况测试结果

分析图表，我们可以发现当CNC加工时间为800秒时，增加RGV没有优化效果。而当CNC加工时间为100秒时，增加RGV的数量对产量有显著的提高。当RGV增加为原来的两倍时，生产速率增加为原来的2.10倍；当RGV增加为原来的三倍时，生产速率增加为原来的3.07倍；当RGV增加为原来的四倍时，生产速率增加为原来的4.13倍。该优化方案在CNC工作时间较长的情况下效果显著。

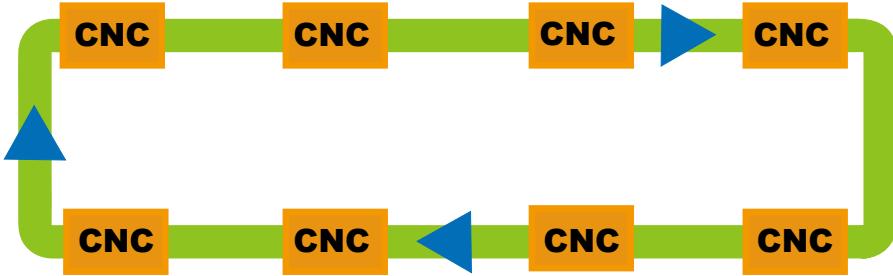


图 16: 环形轨道方案示意图

9.2.2 环形轨道方案

我们对多台RGV情况进行延伸，当CNC数量很多时，多台RGV更容易造成“堵车”。此时一种环形轨道的模型在我们脑中闪现。

环形轨道方案模型如下图所示，矩形代表CNC，三角形代表RGV。我们规定CNC间等距，RGV运行方向固定为顺时针。在该模型下，我们更改算法。我们的程序对RGV进行如下控制：在运动到某一节点时，进行判断，如果该位置的CNC在等待上下料，则进行上下料及清洗操作，否则继续前进。另外，所有在节点停止的RGV都会给上一个节点发送停止信号，如果RGV收到该节点的停止信号，则保持停止直到信号解除。

为了与已有方案形成对比，测试依旧设定与上个方案相同的条件。结果如下表所示：

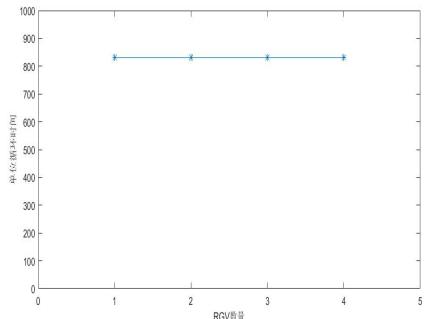


图 17: (a) CNC工作时间为800秒

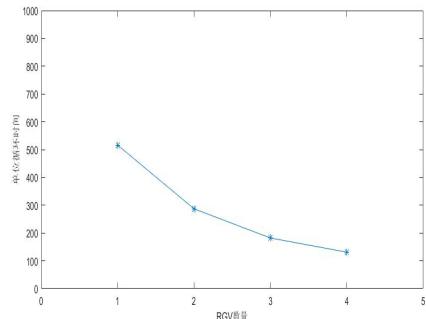


图 18: (b) CNC工作时间为100秒

图 19: 两种情况测试结果

分析图表，我们可以发现当CNC加工时间为800秒时，环形轨道增加RGV依然没有优化效果。而当CNC加工时间为100秒时，增加RGV的数量对产量有显著的提高。当RGV增加为原来的两倍时，生产速率增加为原来的2.17倍；当RGV增加为原来的三倍时，生产速率增加为原来的3.04倍；当RGV增加为原来的四倍时，生产速率增加为原来的4.13倍。

对比于原轨道方案，环形轨道的优化效果在1个RGV时略好一点，因为其节省了从末尾到起始点的少量时间；环形轨道的优化效果在2、3个RGV时略差一点，因为总共只有四个位置，环形轨道的环形收益小于拥堵带来的时间损耗；环形轨道的优化效果在4个RGV时与原轨道相同，因为4个RGV情况RGV不移动，所以生产效率与轨道形式无关。

9.2.3 优化方案总结与提议

通过分析与试验检测，我们发现RGV的增加在CNC工作迅速的情况有明显的优化作用。在RGV相对于CNC数量较少时，引用环形轨道亦可在一定程度增加加工效率。

10 参考文献

- [1] 《基于混合算法的环形轨道RGV系统调度优化研究》江唯, 何非, 童一飞, 李东波
- [2] 《基于TS算法的直线往复2-RGV统调度研究》陈华, 孙启元
- [3] 《环行穿梭车优化调度问题的自学习算法》顾红, 邹平, 徐伟华

11 附录

11.1 单工序的RGV动态调度模型 Python

```
# coding: utf-8
import numpy as np
import random
import copy
import pandas as pd
import math
import matplotlib.pyplot as plt

# RGV_MOVE_ONE = 20
# RGV_MOVE_TWO = 33
# RGV_MOVE_THREE = 46
# CNC_WORK = 560
# RGV_LOAD_FIRST = 28
# RGV_LOAD_SECOND = 31
# RGV_CLEAN = 25
# total_time=3600
# RGV_MOVE_ONE = 23
# RGV_MOVE_TWO = 41
# RGV_MOVE_THREE = 59
# CNC_WORK = 580
# RGV_LOAD_FIRST = 30
# RGV_LOAD_SECOND = 35
# RGV_CLEAN = 30

RGV_MOVE_ONE = 18
RGV_MOVE_TWO = 32
RGV_MOVE_THREE = 46
CNC_WORK = 545
RGV_LOAD_FIRST = 27
RGV_LOAD_SECOND = 32
RGV_CLEAN = 25

Q = 8
finished_goods=0
start_point=1

#global_table
class CNC():
    def __init__(self,num):
        self.number=num
        self.good=0
        self.e_num=0
        self.error=False
        self.clean=False
        self.next_clean=True
        #0 means free, 1 means at work
        self.status=0
        self.time=0 #time needs to finish the work

CNC_array=[] #temp
for i in range(1, 9):
    CNC_array.append(CNC(i))
```

```

# class Good();
#
# Goods_array=[]

class individual():
    def __init__(self, dim):
        self.dim=dim
        self.CNC_array=[]
    # init CNC

    # self.X
def update_time(t):
    global CNC_array
    for i in CNC_array:
        if i.time==0:
            continue
        i.time-=t
        if i.time <0:
            i.time=0
            i.status=0
            i.error=False

class PSO():
    def __init__(self, pN, rN, dim, max_iter):
        self.w = 0.5
        self.c1 = 2
        self.c2 = 2
        self.r1 = 0.6
        self.r2 = 0.3
        self.pN = pN
        self.rN=rN
        self.N=self.pN+self.rN
        self.dim = dim
        self.max_iter = max_iter
        # self.X=[]
        # for i in range(self.pN):
        #     self.X.append(individual(self.dim))
        self.X = np.zeros((self.N, self.dim))
        #self.V = np.zeros((self.pN, self.dim))
        self.V=[]
        for i in range(self.pN):
            self.V.append([])
            for j in range(self.dim):
                k=random.randint(0,8)
                if k!=0:
                    self.V[i].append((j,k))

        # self.CNCs=[]
        # for i in range(self.pN):
        #     self.CNCs=
        self.pbest = np.zeros((self.N, self.dim))
        self.gbest = np.zeros((1, self.dim))
        self.p_fit = np.zeros(self.N)
        self.fit = 1e10
        self.error_list=[]

    def fitness_func(self, x):

```

```

global start_point
b = np.concatenate(([start_point] , x))
length = len(b)
time=0
for i in range(length-1):
    time =time+ self.move(int(b[i]), int(b[i+1])) + self.wait(int(b[i+1])) + self.load(
        int(b[i+1]),i)+self.clean(int(b[i+1])))
#self.fit = time
return time

def move(self, CNC_1, CNC_2):
    global global_location, global_time
    if CNC_1 in [1,3,5,7]:
        a = (CNC_1+1)/2
    else:
        a = CNC_1/2
    if CNC_2 in [1,3,5,7]:
        b = (CNC_2+1)/2
    else:
        b = CNC_2/2
    if abs(a - b) == 0:
        return 0
    elif abs(a - b) == 1:
        #global_time += RGV_MOVE_ONE
        update_time(RGV_MOVE_ONE)
        return RGV_MOVE_ONE
    elif abs(a - b) == 2:
        update_time(RGV_MOVE_TWO)
        #global_time += RGV_MOVE_TWO
        return RGV_MOVE_TWO
    elif abs(a - b) == 3:
        update_time(RGV_MOVE_THREE)
        #global_time += RGV_MOVE_THREE
        return RGV_MOVE_THREE

def load(self, CNC_1,e):
    w=1
    if self.error_list !=[]:
        for i in self.error_list:
            if (e==i[0]):
                CNC_array[CNC_1 - 1].error=True
                CNC_array[CNC_1 - 1].time+=self.error_list[0][1]
                CNC_array[CNC_1-1].next_clean=False
                w=i[2]
    if CNC_1 in [1,3,5,7]:
        update_time(RGV_LOAD_FIRST)
        CNC_array[CNC_1-1].status=1
        CNC_array[CNC_1-1].time+=math.floor(CNC_WORK*w)
        return RGV_LOAD_FIRST
    else:
        update_time(RGV_LOAD_SECOND)
        CNC_array[CNC_1 - 1].status = 1
        CNC_array[CNC_1 - 1].time += math.floor(CNC_WORK*w)
        return RGV_LOAD_SECOND

def clean(self,CNC_1):
    if CNC_array[CNC_1 - 1].clean==False:

```

```

        CNC_array[CNC_1 - 1].clean = True
        return 0
    if CNC_array[CNC_1-1].next_clean==False:
        CNC_array[CNC_1 - 1].next_clean = True
        CNC_array[CNC_1 - 1].clean = False
    update_time(RGV_CLEAN)
    return RGV_CLEAN

def wait(self, CNC_1):
    global global_location, global_time
    C=CNC_array[CNC_1-1]
    if C.status==0:
        return 0
    else:
        tem=C.time
        update_time(C.time)
        return tem

def init_Population(self):
    global CNC_array
    self.fit=1e10
    self.error_list = copy.deepcopy(def_error(self.dim))
    CNC_array = copy.deepcopy(g.CNC_array)
    for i in range(self.N):
        if (i==0):
            for j in range(self.dim):
                self.X[i][j] = j%8 + 1
                #self.X[i][j] = random.randint(1, 8)
        # elif (i==2):
        #     for j in range(8):
        #         self.X[i][j] = j % 8 + 1
        #     for j in range(8,14):
        #         self.X[i][j] =j-5
        #     for j in range(14,16):
        #         self.X[i][j] =j-11

        else:
            for j in range(self.dim):
                self.X[i][j] = random.randint(1,8)
                #self.X[i][j] = j+1
                #self.V[i][j] = random.uniform(0, 1) ???
            self.pbest[i] = copy.deepcopy(self.X[i])
            tmp = self.fitness_func(self.X[i])
            self.p_fit[i] = tmp
            if (tmp < self.fit):
                self.fit = tmp
                self.gbest = copy.deepcopy(self.X[i])

```

```

def iterator(self):
    global CNC_array,g
    fitness = []
    for t in range(self.max_iter):

        for i in range(self.N):
            temp = self.fitness_func(self.X[i])
            if (temp < self.p_fit[i]):

```

```

        self.p_fit[i] = temp
        self.pbest[i] = copy.deepcopy(self.X[i])
        if (self.p_fit[i] < self.fit):
            self.gbest = copy.deepcopy(self.X[i])
            self.fit = self.p_fit[i]
    for i in range(self.pN):
        self.V[i] = self.pick(self.w, self.V[i]) + self.pick(self.r1,(self.minus(self
                           .pbest[i] , self.X[i])))) + \
                    self.pick(self.r2,(self.minus(self.gbest, self.X[i]))))
        self.X[i] = self.change(self.X[i] , self.V[i])
    for i in range(self.pN,self.N):
        for j in range(self.dim):
            self.X[i][j] = random.randint(1,8)
    fitness.append(self.fit)
    #print(self.fit,self.gbest)
    print(self.fit, self.gbest, self.error_list)
    g.update_global(self.gbest, self.error_list)
    return fitness

def minus(self,a,b): #self.pbest[i] - self.X[i]
    result=[]
    for i in range(self.dim):
        if (a[i]!=b[i]):
            result.append((i,a[i]))
    return result

def pick(self,p,v):
    result=[]
    for i in v:
        r=random.random()
        if (r<p):
            result.append(i)
    return result

def change(self,x,v):
    for i in v:
        x[i[0]]=i[1]
    return x

def def_error(times):
    result=[]
    for i in range(times):
        p=random.random()
        if (p<=0.01):
            t=math.floor((random.random()*10+10)*60)
            w=random.random()
            result.append([i,t,w])
    return result
    #return []

class global_info():
    def __init__(self):
        self.good_num=0
        self.error_num=0
        self.global_time = 0
        self.global_location = 1
        #self.finished_goods = 0
        self.CNC_array = []
        for i in range(1, 9):

```

```

        self.CNC_array.append(CNC(i))
        self.table = []
        self.error_table = []
        self.error_list = []

    def update_global(self, gb, e):
        global start_point
        self.error_list = e;
        b = np.concatenate(([start_point], gb))
        length = len(b)
        for i in range(length - 1):
            self.move(int(b[i]), int(b[i + 1]))
            self.wait(int(b[i + 1]))
            self.load(int(b[i + 1]), i)
            self.clean(int(b[i + 1]))
        # update start_point
        start_point = gb[-1]

    def update_time(self, t):
        self.global_time += t
        for i in self.CNC_array:
            if i.time == 0:
                continue
            i.time -= t
            if i.time < 0:
                i.time = 0
                i.status = 0
                i.error = False

    def move(self, CNC_1, CNC_2):
        if CNC_1 in [1, 3, 5, 7]:
            a = (CNC_1 + 1) / 2
        else:
            a = CNC_1 / 2
        if CNC_2 in [1, 3, 5, 7]:
            b = (CNC_2 + 1) / 2
        else:
            b = CNC_2 / 2
        if abs(a - b) == 0:
            return 0
        elif abs(a - b) == 1:
            #global_time += RGV_MOVE_ONE
            self.update_time(RGV_MOVE_ONE)
            return RGV_MOVE_ONE
        elif abs(a - b) == 2:
            self.update_time(RGV_MOVE_TWO)
            #global_time += RGV_MOVE_TWO
            return RGV_MOVE_TWO
        elif abs(a - b) == 3:
            self.update_time(RGV_MOVE_THREE)
            #global_time += RGV_MOVE_THREE
            return RGV_MOVE_THREE

    def load(self, CNC_1, e):
        w = 1
        C = self.CNC_array[CNC_1 - 1]
        if C.good != 0:
            self.table[C.good - 1][-1] = self.global_time

```

```

#C.good=0
self.good_num += 1
C.good = self.good_num
self.table.append([self.good_num, CNC_1, self.global_time,-1])
if self.error_list != []:
    for i in self.error_list:
        if (e==i[0]):
            self.error_num+=1
            CNC_array[CNC_1 - 1].error=True
            CNC_array[CNC_1 - 1].time+=i[1]
            CNC_array[CNC_1 - 1].next_clean = False
            #CNC_array[CNC_1 - 1].good=0
            w=i[2]

if CNC_1 in [1, 3, 5, 7]:
    self.update_time(RGV_LOAD_FIRST)
    self.CNC_array[CNC_1-1].status=1
    if CNC_array[CNC_1 - 1].error==True:
        self.error_table.append([self.good_num, CNC_1, self.global_time+math.floor(
            CNC_WORK*w), \
            self.global_time+math.floor(CNC_WORK*w+i[1])])
    self.CNC_array[CNC_1-1].time+=math.floor(CNC_WORK*w)
    return RGV_LOAD_FIRST
else:
    self.update_time(RGV_LOAD_SECOND)
    self.CNC_array[CNC_1 - 1].status = 1
    if CNC_array[CNC_1 - 1].error==True:
        self.error_table.append([self.good_num, CNC_1, self.global_time+math.floor(
            CNC_WORK*w), \
            self.global_time+math.floor(CNC_WORK*w+i[1])])
    self.CNC_array[CNC_1 - 1].time +=math.floor(CNC_WORK*w)
    return RGV_LOAD_SECOND

def clean(self,CNC_1):
    if CNC_array[CNC_1 - 1].clean==False:
        CNC_array[CNC_1 - 1].clean = True
    return 0
    if CNC_array[CNC_1-1].next_clean==False:
        CNC_array[CNC_1 - 1].next_clean = True
        CNC_array[CNC_1 - 1].clean = False
    update_time(RGV_CLEAN)
    return RGV_CLEAN

def wait(self, CNC_1):
    C=self.CNC_array[CNC_1-1]
    if C.status==0:
        return 0
    else:
        tem=C.time
        self.update_time(C.time)
        return tem

g=global_info()
my_pso = PSO(pN=200, rN=200,dim=16, max_iter=100)

while (g.global_time<28800):
    my_pso.init_Population()
    my_pso.iterator()

```

```

# while len(g.table[-1]) !=4:
#     g.table.pop()

print("-----")
print(g.table)
print(g.error_table)

np_data = np.array(g.table)

pd_data.to_csv('case3_result1_3.csv')

np_data_e = np.array(g.error_table)

pd_data.to_csv('case3_result1_3_error.csv')

# plt.figure(1)
# plt.title("Figure1")
# plt.xlabel("iterators", size=14)
# plt.ylabel("fitness", size=14)
# t = np.array([t for t in range(0, 100)])
# fitness = np.array(fitness)
# plt.plot(t, fitness, color='b', linewidth=3)
# plt.show()

```

11.2 双工序的RGV动态调度模型 Python

```

import numpy as np
import matplotlib.pyplot as plt
import math
import copy
import csv
import pandas as pd

# RGV_MOVE_ONE = 20
# RGV_MOVE_TWO = 33
# RGV_MOVE_THREE = 46
# CNC_FIRST = 400
# CNC_SECOND = 378
# RGV_LOAD_FIRST = 28
# RGV_LOAD_SECOND = 31
# RGV_CLEAN = 25
# global_time = 840
# global_location = 1
# Q = 8

# RGV_MOVE_ONE = 23
# RGV_MOVE_TWO = 41
# RGV_MOVE_THREE = 59

```

```

# CNC_FIRST = 280
# CNC_SECOND = 500
# RGV_LOAD_FIRST = 30
# RGV_LOAD_SECOND = 35
# RGV_CLEAN = 30
# global_time = 840
# global_location = 1
# Q = 8

RGV_MOVE_ONE = 18
RGV_MOVE_TWO = 32
RGV_MOVE_THREE = 46
CNC_FIRST = 455
CNC_SECOND = 182
RGV_LOAD_FIRST = 27
RGV_LOAD_SECOND = 32
RGV_CLEAN = 25
global_time = 942
global_location = 1
Q = 8
ERROR = 0.01

class CNC:
    def __init__(self, number, form, product_number = 0):
        self.product_number = product_number
        self.number = number
        self.form = form
        self.start = 0
        self.cnc_start = 0
        self.cnc_end = 0
        self.new = 0
        self.previuos = 0
        self.broke = False
        self.broke_start = 0
    def __str__(self):
        return str(self.broke)

# def fitness(x):
#     return x*np.sin(10*np.pi*x) + 2

First_1 = CNC(1, 'FIRST', 5)
First_1.previuos = 5
First_2 = CNC(2, 'FIRST', 6)
First_2.previuos = 6
First_3 = CNC(3, 'FIRST', 7)
First_3.previuos = 7
First_4 = CNC(4, 'FIRST', 8)
First_4.previuos = 8
First = [First_1, First_2, First_3, First_4]

Second_1 = CNC(11, 'SECOND', 1)
Second_2 = CNC(12, 'SECOND', 2)
Second_3 = CNC(13, 'SECOND', 3)
Second_4 = CNC(14, 'SECOND', 4)
Second = [Second_1, Second_2, Second_3, Second_4]

info = []

```

```

error_log = []
pe_info = []
product_count = 8
dic = {'1':1, '2':3, '3':5, '4':7, '11': 2, '12':4, '13':6, '14':8}

class indivdual:

    def __init__(self):
        global global_location, global_time
        global First
        global Second
        self.x = []
        self.real_key = False
        self.First = copy.deepcopy(First)
        self.Second = copy.deepcopy(Second)
        self.global_time = copy.deepcopy(global_time)
        self.information = []
        self.count = 0
        self.dic = {'1':1, '2':3, '3':5, '4':7, '11': 2, '12':4, '13':6, '14':8}
        self.a = []
        self.b = []
        for cnc in range(len(self.First)):
            if self.First[cnc].broke == False or (self.global_time - self.First[cnc].broke_start) >=600:
                self.a.append(self.First[cnc].number)
                self.First[cnc].broke == False
            else:
                pass
        for cnc in range(len(self.Second)):
            if self.Second[cnc].broke == False or (self.global_time - self.Second[cnc].broke_start) >=600:
                self.b.append(self.Second[cnc].number)
                self.Second[cnc].broke == False
            else:
                pass
        for i in range(8):
            if i < 4:
                m = np.random.choice(self.a, 1)[0]
                self.x.append(m)
            else:
                n = np.random.choice(self.b, 1)[0]
                self.x.append(n)
        #print(self.x)
        self.fit = self.fitness()
    def initial(self):
        global First, Second, global_time
        self.First = copy.deepcopy(First)
        self.Second = copy.deepcopy(Second)
        self.global_time = copy.deepcopy(global_time)
    def fitness(self):
        global global_location
        time = self.move(self.First[global_location-1], self.First[self.x[0]-1]) + self.wait(
            self.First[self.x[0] - 1]) + self.load(
            self.First[self.x[0]-1])
        #print(time)
        time += self.move(self.First[self.x[0]-1], self.Second[self.x[4] - 11]) + self.wait(
            self.Second[self.x[4] - 11]) + self.load(
            self.Second[self.x[4] - 11]) +
            self.clean()

```

```

#print(time)
time += self.move(self.Second[self.x[4] - 11], self.First[self.x[1]-1] ) + self.wait(
    self.First[self.x[1]-1]) + self.load(
    self.First[self.x[1]-1])

#print(time)
time += self.move(self.First[self.x[1]-1], self.Second[self.x[5] - 11]) + self.wait(
    self.Second[self.x[5] - 11]) + self.
load(self.Second[self.x[5] - 11]) +
self.clean()

#print(time)
time += self.move(self.Second[self.x[5] - 11], self.First[self.x[2]-1]) + self.wait(
    self.First[self.x[2]-1]) + self.load(
    self.First[self.x[2]-1])

#print(time)
time += self.move(self.First[self.x[2]-1], self.Second[self.x[6] - 11]) + self.wait(
    self.Second[self.x[6] - 11]) + self.
load(self.Second[self.x[6] - 11]) +
self.clean()

#print(time)
time += self.move(self.Second[self.x[6] - 11], self.First[self.x[3]-1]) + self.wait(
    self.First[self.x[3]-1]) + self.load(
    self.First[self.x[3]-1])

#print(time)
time += self.move(self.First[self.x[3]-1], self.Second[self.x[7] - 11]) + self.wait(
    self.Second[self.x[7] - 11]) + self.
load(self.Second[self.x[7] - 11]) +
self.clean()

#print(time)
return time

def move(self, CNC_1, CNC_2):
    if CNC_1.number > 10:
        a = CNC_1.number - 10
    else:
        a = CNC_1.number
    if CNC_2.number > 10:
        b = CNC_2.number - 10
    else:
        b = CNC_2.number
    if abs(a - b) == 0:
        return 0
    elif abs(a - b) == 1:
        self.global_time += RGV_MOVE_ONE
        return RGV_MOVE_ONE
    elif abs(a - b) == 2:
        self.global_time += RGV_MOVE_TWO
        return RGV_MOVE_TWO
    elif abs(a - b) == 3:
        self.global_time += RGV_MOVE_THREE
        return RGV_MOVE_THREE

def load(self, CNC_1):
    global error_log
    global product_count, ERROR
    if CNC_1.form == 'FIRST':
        if self.real_key:
            product_count += 1
            CNC_1.product_number = product_count
            CNC_1.previoos = CNC_1.product_number - 1
            CNC_1.cnc_start = self.global_time
            point = np.random.random()

```

```

        if (self.global_time - CNC_1.broke_start) >=600:
            CNC_1.broke = False
        if point < ERROR:
            CNC_1.broke = True
            CNC_1.broke_start = self.global_time
            error_log.append([CNC_1.product_number, self.dic[str(CNC_1.number)], CNC_1.
                broke_start, CNC_1.
                broke_start + np.random.
                randint(750,810)])
        self.global_time += RGV_LOAD_FIRST
        return RGV_LOAD_FIRST
    elif CNC_1.form == 'SECOND':
        if self.real_key:
            p = self.x.index(CNC_1.number)
            f = self.x[p-4]
            CNC_1.product_number = First[f-1].previuos
            CNC_1.cnc_start = self.global_time
            point = np.random.random()
            if (self.global_time - CNC_1.broke_start) >=600:
                CNC_1.broke = False
            if point < ERROR:
                CNC_1.broke = True
                CNC_1.broke_start = self.global_time
                error_log.append([CNC_1.product_number, self.dic[str(CNC_1.number)], CNC_1.
                    broke_start, CNC_1.
                    broke_start + np.random.
                    randint(550,610)])
        self.global_time += RGV_LOAD_SECOND
        return RGV_LOAD_SECOND
    def clean(self):
        self.global_time += RGV_CLEAN
        return RGV_CLEAN
    def wait(self, CNC_1):
        global info
        if CNC_1.new == 0:
            if self.real_key:
                if CNC_1.form == 'FIRST':
                    CNC_1.cnc_start = self.global_time
                elif CNC_1.form == 'SECOND':
                    CNC_1.cnc_start = self.global_time
                    CNC_1.cnc_end = self.global_time
                    info.append([CNC_1.product_number, CNC_1.form, self.dic[str(CNC_1.number)],
                        CNC_1.cnc_start, CNC_1.cnc_end])
        CNC_1.start = self.global_time + RGV_LOAD_FIRST
        CNC_1.new = 1
        return 0

```

```

else:
    if CNC_1.form == 'FIRST':
        a = CNC_FIRST - (self.global_time - CNC_1.start)
        if a <=0:
            if self.real_key:
                CNC_1.cnc_end = self.global_time
                info.append([CNC_1.product_number, CNC_1.form, self.dic[str(CNC_1.
                    number)], CNC_1.
                    cnc_start, CNC_1.

```

```

        cnc_end])
    CNC_1.start = self.global_time + RGV_LOAD_FIRST
    return 0
else:
    self.global_time += a
    if self.real_key:
        CNC_1.cnc_end = self.global_time
        info.append([CNC_1.product_number, CNC_1.form, self.dic[str(CNC_1.
number)], CNC_1.
cnc_start, CNC_1.
cnc_end])
    CNC_1.start = self.global_time + RGV_LOAD_FIRST
    return a
if CNC_1.form == 'SECOND':
    a = CNC_FIRST - (self.global_time - CNC_1.start)
    if a <=0:
        if self.real_key:
            CNC_1.cnc_end = self.global_time
            info.append([CNC_1.product_number, CNC_1.form, self.dic[str(CNC_1.
number)], CNC_1.
cnc_start, CNC_1.
cnc_end])
    CNC_1.start = self.global_time
    return 0
else:
    self.global_time += a
    if self.real_key:
        CNC_1.cnc_end = self.global_time
        info.append([CNC_1.product_number, CNC_1.form, self.dic[str(CNC_1.
number)], CNC_1.
cnc_start, CNC_1.
cnc_end])
    CNC_1.start = self.global_time
    return a

def __eq__(self, other):
    self.x = other.x
    self.fitness = other.fitness
def __str__(self):
    return str(self.x) + ' ' + str(self.fitness)

def initPopulation(pop, N):
    for i in range(N):
        ind = individual()
        pop.append(ind)
    return pop
# def foo(temp):
#     su = 0
#     pointer = 0
#     seed = np.random.random()
#     for i in range(len(temp)):
#         su += temp[i]
#         if seed <= pointer:
#             pointer = i
#     return pointer

```

```

# def selection(N, pop):
#
#     temp = []
#     for i in pop:
#         temp.append(1/i.fit)
#     a = sum(temp)
#     for i in range(len(temp)):
#         temp[i] = temp[i]/a
#     b = sum(temp)
#     for i in range(len(temp)):
#         temp[i] = temp[i]/b
#     pointer1 = foo(temp)
#     pointer2 = foo(temp)

#     return [pointer1, pointer2]

def selection(N):
    return np.random.choice(N, 2)

def crossover(parent_1, parent_2):
    global global_time, First, Second
    child1, child2 = individual(), individual()
    seed = np.random.randint(1,5)
    parent1 = copy.deepcopy(parent_1)
    parent2 = copy.deepcopy(parent_2)
    temp1 = parent1.x[seed - 1]
    temp2 = parent1.x[seed + 3]
    parent1.x[seed - 1] = parent2.x[seed - 1]
    parent1.x[seed + 3] = parent2.x[seed + 3]
    parent2.x[seed - 1] = temp1
    parent2.x[seed + 3] = temp2
    child1.x = parent1.x
    child2.x = parent2.x
    child1.global_time = global_time
    child2.global_time = global_time
    child1.First = copy.deepcopy(First)
    child1.Second = copy.deepcopy(Second)
    child2.First = copy.deepcopy(First)
    child2.Second = copy.deepcopy(Second)
    child1.fit = child1.fitness()
    child2.fit = child2.fitness()
    return child1, child2

def mutation(pop):
    global global_time, First, Second

    ind = np.random.choice(pop)
    a = []
    b = []

    for cnc in range(len(First)):
        if First[cnc].broke == False:
            a.append(First[cnc].number)
        else:
            pass
    for cnc in range(len(Second)):
        if Second[cnc].broke == False:
            b.append(Second[cnc].number)

```

```

        else:
            pass
seed = np.random.randint(1,5)
ind.x[seed - 1] = np.random.choice(a, 1)[0]
ind.x[seed + 3] = np.random.choice(b, 1)[0]
ind.global_time = copy.deepcopy(global_time)
ind.First = copy.deepcopy(First)
ind.Second = copy.deepcopy(Second)
ind.fit = ind.fitness()

def implement():

    N = 200

    POP = []

    iter_N = 8000

    POP = initPopulation(POP, N)

    for it in range(iter_N):
        a, b = selection(N)
        if np.random.random() < 0.8:
            child1, child2 = crossover(POP[a], POP[b])
            new = sorted([POP[a], POP[b], child1, child2], key=lambda ind: ind.fit, reverse=False)
            POP[a], POP[b] = new[0], new[1]

        if np.random.random() < 0.4:
            mutation(POP)

    POP.sort(key=lambda ind: ind.fit, reverse=False)
    global global_location, global_time, First, Second
    # POP[0].global_time = global_time
    # POP[0].First = copy.deepcopy(First)
    # POP[0].Second = copy.deepcopy(Second)
    # POP[0].fit = POP[0].fitness()
    temp = individual()
    temp.x = POP[0].x
    temp.initial()
    temp.real_key = True
    temp.fit = temp.fitness()
    global_location = POP[0].x[3]
    global_time = POP[0].global_time
    First = copy.deepcopy(temp.First)
    for i in First:
        print(i)
    Second = copy.deepcopy(temp.Second)
    return temp

# pop = implement()

# print(pop.fit)
# print(pop.x)
# print(global_time)
# print(info)
def main():

```

```

global pe_info
for i in range(70):
    pop = implement()
    print(pop.fit)
    print(pop.x)
    pe_info.append([i, pop.fit, pop.x])
    print(global_time)
main()
info.sort(key = lambda ind: ind[3])
csvfile = open('csvtest.csv', 'w')
writer = csv.writer(csvfile)
data = []
for i in range(800):
    data.append([0,0,0,0,0,0,0])
for i in range(len(info)):
    if info[i][1] == 'FIRST':
        data[info[i][0]][0] = info[i][0]
        data[info[i][0]][1] = info[i][2]
        data[info[i][0]][2] = info[i][3]
        data[info[i][0]][3] = info[i][4]
    elif info[i][1] == 'SECOND':
        data[info[i][0]][0] = info[i][0]
        data[info[i][0]][4] = info[i][2]
        data[info[i][0]][5] = info[i][3]
        data[info[i][0]][6] = info[i][4]
for i in range(len(data)):
    writer.writerow(data[i])

np_data = np.array(data)

if not (error_log == []):

```