同济大学 2019 年数学建模 B 题

动车组检修

基于遗传算法的动车组检修问题求解

摘要

流水车间调度问题，是许多实际流水线生产调度问题的简化模型，它无论是在离散制造工业还是在流程工业中都有较为广泛的应用，因此，对该问题的研究具有非常重要的工程价值。

在动车组运用所检修作业调度中，针对更为实际的检修调度场景，精确掌握检修调度的约束条件，建立更可靠的求解最优化调度方案的模型，就能制定和采取更有效的管理和调节措施。对于以提高检修工作效率为主要目的的动车运用所，我们的目标函数即是该批次检修车辆最小完工时间。

而针对不同的情况，我们计算目标函数时其对应的约束条件也不同，本文分别讨论了同样动车不同时刻到达、不同动车不同时刻到达以及作业车间加工调度（特定车辆需要特定加工工序）这三种情况，我们利用类比以及逐层添加约束的思想，分别建立了模型。

对遗传算法与流水调度车间问题进行耦合性对比，发现利用遗传算法中适应度函数的倒数来描述目标函数是可行的。通过不同的约束条件对不同情况的检修过程进行优先级的排序（约束条件：到站间隔时间，每道工序的检修时间，检修的工序数量），再对这些种群中的个体进行选择、交叉、变异，并迭代生成新种群，在满足终止条件时，输出最优解，即问题的所求解。

最后，我们对我们建立的模型进行了改进、评估和反思，优化了迭代效率，提高了模型的准确性和稳定性。

关键词：混合流水车间调度，遗传算法，概率模型，选择算子，交叉算子

目录

一、问题重述

1. 问题背景

全国现已建成超过 50 个动车组运用所，复制对动车进行检修、养护等工作。动车组的检修需要按工序顺序依次进行。每个工序拥有的作业车间和需要花费的时间各不相同。动车组序按照一定的顺序检修，完成一个检修工序后驶入下一个有空闲位置的车间进行下一个检修工序，若下一个工序所有车间都处于占用状态，则动车组需要在上一个车间中等待。根据动车行驶情况不同，动车组被划分为多个不同的等级，不同的等级对应不同的工序组合，不同工序的检修时间也有所不同。

如何建立数学模型，有效缩短动车检修、养护的时间，是动车运用所提高效率，保证工作质量的研究重点。

2. 目标任务

根据已提供的数据资料，解决以下问题：

（1） 找出在相同工序不同车间耗费时间相同，且与动车组类型无关的条件下，动车运用所在 12 小时内每隔 15 分钟迎来一辆待检修动车。按照给定车间设置，维修完所有的动车组共需耗费的最短时间。

（2） 在不同类型的动车组每个工序花费时间不一样的条件下，根据列车到站时间信息表，找出维修完所有动车组需要耗费的总最短时间。

（3） 在不同类型的动车检修等级不同，不同检修等级对应的工序组合不同条件下，根据列车检修等级信息与到站时刻信息，给出维修完所有动车组需要耗费的最短总时间。

二、问题分析

动车组检修调度问题就是利用动车组运用所的车间资源对动车进行检修的过程，作业调度则是对检修作业进行有效排序，使某个目标函数最小。

问题一：关于同样动车不同时刻到达检修站点进行检修的最优分配问题。

由于是同样的动车，因此对于相同的工序类别，即 a、b、c，对应的检修时间全部都是相同的，此题在这个角度上来讲具有对称性，它们优先级的差异无非就是来自下一辆车滞后于上一辆车十五分钟到达。（每道工序不同机器具有等价性）那么我们接下来就来讨论如何分配来促使时间最少（最优解）。

根据问题的叙述，对于每种车辆的类别都相同的情况，调整车辆检修顺序不会导致结果的变化，因此，为了节约时间，使得完工时间尽量最少，我们采取先驶入先检修的办法。

即第某辆车如果到达了检修站点，并且第一个工序还剩有空余机器，那么便立刻让它进入检修程序。

因此整个检修过程变为了：第一辆车进入第一道工序第一个机器，等待十五分钟后，第二辆车进入第一道工序第二个机器，再经过十五分钟后第三辆车进入第一道工序第三个机器，由于第一道工序已被填满，再等待十五分钟后车辆无法进入检修程序，需等待下一个时刻才能进入，此后若有类似情况（车辆滞留），即等待第一个工序有机器空闲出来即可，由此，对整个过程进行分析计算，我们可以得到第一题的最优解。

问题二：关于不同动车不同时刻到达检修站点进行检修的最优分配问题。

该问题与第一问比较增加了两个约束条件。

第一， 到站的间隔时间并非固定，而是随着车次的变化而变化。

第二， 不同的动车在相同的工序类别中(如 a、b、c)对应的检修时间是不同的。

所有调度的可能性取决于不同车同时进入一轮检修环节时谁先进入，也就是优先级。可以通过穷举所有优先级的排列组合也就是所有调度的可能性来求解动车组检修的最优分配问题。但是穷举方法只适用于小规模数据量的求解问题，并且求解的效率太低。因此不采用这种方法，而是采用遗传算法这种智能搜索算法进行求解。

在计算他们的优先级时需要考虑不同的滞后时间以及不同的检修时间。

下面我们简单来讨论一下优先级与变量之间的关系。

很明显，当车型相同时，我们采用的是先驶入先检修的方法，其实际上是对不同车次的优先级进行了排序，即滞后时间越短，优先级越高，因此对于该题，即便是滞后间隔时间并不同，但是二者依旧是呈负相关的关系。

那么同理，检修所需要的时间越短，相应的优先级也就越高，因此我们在建立模型时会充分考虑到这一点。

问题三： 关于不同动车不同时刻到达站点后需要检修的工序数量也不同的最优分配问题。

前述为流水加工调度问题，那么该题则是作业车间加工调度问题（即不同工件需要特定的加工工序。）我们依旧是采用排列优先级的思想。

但是现在的优先级很明显又需要加入新的参量（工序数），很明显如果两辆不同的车加工的工序数不一样，例如 A 车需要三个工序，B 车需要四个工序，C 车需要五个工序（假设所有的车里面 C 车需要的工序数最多），我们只需为 A 车添加两个工序，B 车添加一个工序即可。A 车的第四第五个工序所需的时间为 0，B 车第五个工序所需的时间为 0，）那么这个问题就转化为了第二个问题，解题的模板将会完全一致，只需修改数据即可。

三、模型假设与符号说明

1. 模型假设

（1）假设同一工序中所有车间都完全相同。
（2）加工过程中，排除因各种原因出现的检修中断事件。
（3）每列动车可以在某一工序中任意一间车间进行检修。

（4）每个车间至多同时检修一列动车。
（5）在完成某道工序后，在转运到下一道工序的时间忽略不计。

2．符号说明

符号	释义
n	需要检修的动车数量
c	每列动车需要进行的检修工序数
m	最大并行机器数
m_k	第 k 阶段的并行机器数
$o(k,j)\ \ (k\in c, j\in n)$	优先级为 j 的动车的第 k 道工序
$p(k,j)\ \ (k\in c, j\in n)$	优先级为 j 的第 k 道工序所需要的检修时间
$i_k\ (i\in m)$	第 k 道工序的第 i 个车间
$C(i_k,j)$	优先级为 j 的动车在第 k 道工序中 i 车间里完工的时间
N	每代种群的个体数
$F_i\ (i=1,2,...,N)$	个体的适应度值
G	进化的迭代次数

四、模型建立与求解

1. 模型分析

根据假设和题目所给条件，列车的检修过程可以看作建立遗传算法解决混合流水车间调度问题。

遗传算法（Genetic Algorithm）是一种模拟自然选择和生物进化过程的智能优化算法。在自然界中，自从达尔文提出“优胜劣汰，适者生存”物种进化理论之后，研究学者对生物进化的过程进行了长久而又深远的研究。物种通过母代的繁衍形成新的下一代个体，新一代个体中，大多数个体由于发生染色体交叉过程会与母代类似，少数个体由于发生了变异则与母代不同。大自然作为自然选择的执行者，在生存资源和外界环境的变化、个体不断进行竞争的过程中，将适应能力强的个体留下，而淘汰适应能力差的个体，这种自然选择的过程为人类提供了一种全新的解决问题的方式。

遗传算法的基本思想生物的进化是通过染色体来实现的，染色体上有着许多

控制生物性状的基因，这些基因会在遗传过程中随着染色体的交叉进行重新组合，同时会以一定概率发生变异。遗传算法的基本思路与此类似，可以将待优化问题的求解看作生物努力适应环境的过程，问题的解对应生物种群中的个体，算法的搜索便是种群一代代进化最终形成稳定物种的过程。

混合流水车间调度问题简介混合流水车间调度问题（Hybrid Flow Shop Scheduling Problem, HFSSP）也称为柔性流水车间调度问题，是经典流水车间调度的推广。它综合了经典流水车间和并行机两种调度的特点，符合实际生产的要求，具有很高的研究价值和应用背景。

下图简单表示了 HFSSP 问题，其中假设有c加工阶段，每个阶段i有

$c_i\left(i=1,2,.....m\right)$台机器。

HFSSP 考虑在一条流水线上进行生产。n个工件在包含c个阶段（机器中心）的流水线上进行加工。每个工件都要依次通过每个阶段。每个阶段至少有一台加工机器并且至少有一个阶段包含多台并行机器。

2. 模型建立

将进入动车运用所的所有车按到达顺序从小到大编号，将这些编号的一种排列组合看作生物的一个个体。足够多的不同排列组合的个体组成一个生物种群。这个生物种群中的个体模拟自然选择和生物进化过程，自然繁衍，优胜劣汰，适者生存。

遗传算法的结构框架可以简述如下：

1、初始化：依据每个种群的特征随机生成第一代种群的全部个体；

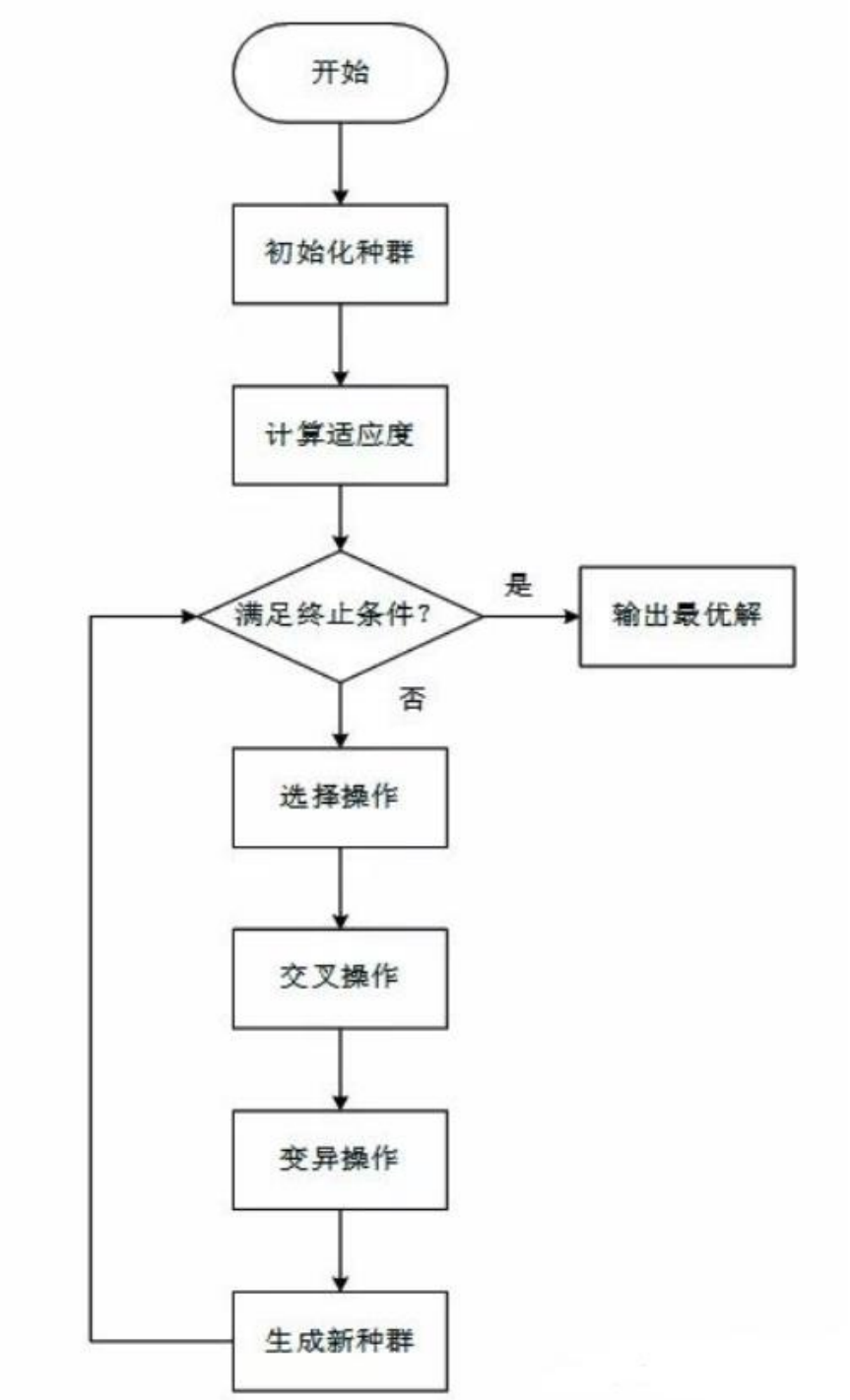
2、求个体适应度：计算每个个体的适应度；

3、选择过程：依据一定的选择规范，选出一部分优秀个体参与交叉和变异操作；

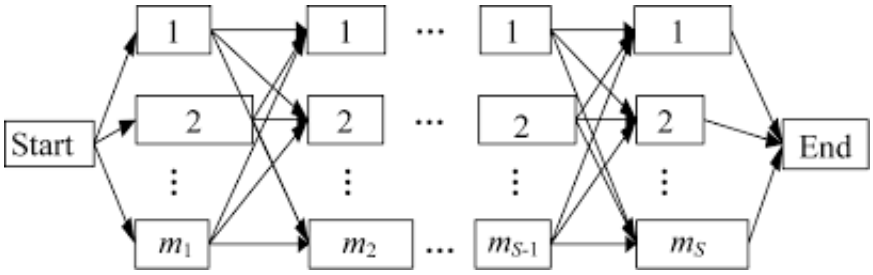
4、交叉过程：群体中两两配对，交换部分染色体基因，完成交叉操作；

5、变异过程：随机改变个体中的部分基因，来实现变异操作；

6、终止判断：若新一代种群满足终止条件，停止算法迭代，记录此时的最优解为问题的最优解；否则，迭代次数加 1，返回步骤 2；附遗传算法的算法流程图：



混合流水车间调度问题的模型示意图如下：



本题的模型思想基于混合流水车间调度问题简介混合流水车间调度问题，但不同点在于以下几点：

（1）不同车次到站时间不同，开始检修的时间也存在时间差。

（2）不同工序中的车间数量并不相等。

（3）不同的车次需要经过的检修工序不完全相同。

已知各工件的加工时间，优化目标是如何确定动车的检修顺序以及每阶段动车在车间的分配情况，使得最大完工时间极小化。

2.1 目标函数的确定

在调度的实际生产过程中，不同的部门对各个目标有不同的侧重，例如制造部门希望用最低的生产损失实现最高的销售目标，销售部门希望在与客户签订的交货期内，完成产品生产，而产品检修方面则希望最高效利用每一个工序的所有机器在尽可能短的时间完成产品的检修。

最大完工时间是指动车检修中所有的动车全部检修完成后的时间，包括动车的最后一道检修工序，通常情况下，该指标是评价车间调度问题的基本指标，也是最能代表企业生产效率的指标，本文采用最大完工时间作为唯一目标函数，它的函数表示为：

$$Makespan = \max C\left(i_k, j\right)$$

当最大完工时间取得最小值时，即得到最小生产周期时，表示这条加工路线时间最短，目标函数的值就取得极小值，动车运用所的检修效率也就取得极大值。

2.2 约束条件

$$C\left(i_1,1\right) = p\left(1,1\right) \tag{1}$$

$$C\left(i_k,1\right) = C\left(i_{k-1},1\right) + p\left(k,1\right) \tag{2}$$

$$C\left(i_1, j\right) = C\left(i_1, j-1\right) + p\left(1, j\right) \tag{3}$$

$$C\left(i_k, j\right) = \max\left\{C\left(i_{k-1}, j\right), C\left(i_k, j-1\right)\right\} + p\left(k, j\right) \tag{4}$$

(1)式表示第 1 个动车第 1 道工序的检修完成时间等于该动车在第一阶段的完工时间；

(2)式表示第 1 个动车第 k 道工序检修完成的时间等于该动车紧前工序的检修完成时间加上当前工序的检修时间；

(3)式表示动车 j 第 1 道工序的完工时间，等于同一车间里紧前动车第 1 道工序的完工时间加上动车 j 第 1 道工序的检修时间；

(4)式表示动车 j 在阶段 k 的检修完成时间，等于动车 j 紧前工序的检修完成

时间或同一车间紧前动车 $j-1$ 的检修完成时间中的最大值加上动车 j 在阶段 k 的检修时间。

3.算法设计

3.1 基因编码

种群中每一个个体有唯一的线性基因。我们将个体的基因定义为，动车组集合中不同动车在同时可以进入下一检修轮次时谁先进入的优先顺序。采用随机全排列的方法生成 n 辆动车的不同顺序结构。这一顺序结构（基因）代表了处理的优先级，基因编码从左到右优先级依次下降。如编为如 1 4 6 5 2 7 3 的基因编码（以七辆动车组为例）。4 号动车组检修的优先级优先于六号动车组。

3.2 种群初始化

将 N 个随机排列生成的基因作为初始种群。随机全排列实现方法采用先生成顺序排列的基因然后生成基因区间范围内的两个随机数。让位于这两个随机数对应位置的基因互换。通过循环若干次互换过程来实现随机全排列的基因。从而得到不同基因的个体作为遗传算法种群的初始化。

2.1 适应值函数

在遗传算法中，适应度越高，被选择的几率就越大。在本次设计中，将适应度函数表示成目标函数倒数的形式，即

$$F = \frac{1}{Makespan}$$

其中 $Makespan$ 表示目标函数，即最小生产周期，这样，当生产周期最小时，适应度函数值最大，就表示这条加工路线时间最短。

3.3 选择操作

对于种群中需要进行杂交的物种选择方法有很多，而且选择一种合适的选择策略对于遗传算法整体性能的影响将是很大的。如果一个选择算法选择多样性降低，便会导致种群过早的收敛到局部最优点而不是我们想要的全局最优点，也就是所谓的”早熟”。而选择策略过于发散则会导致算法难以收敛到最优点。因此在这两点中我们需要进行平衡才能使遗传算法以一种高效的方式收敛到全局最优点。

遗传算法的常见选择操作有四种方法，分别是：

1. 轮盘赌法

此轮盘赌选择策略，是最基本的选择策略之一，种群中的个体被选中的概率与个体相应的适应度函数的值成正比。我们需要将种群中所有个体的适应度值进行累加然后归一化，最终通过随机数对随机数落在的区域对应的个体进行选取，类似赌场里面的旋转的轮盘。

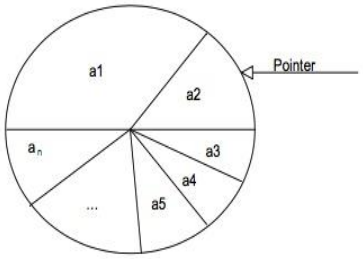


Fig. 2. Roulette wheel selection [22]

采用轮盘赌的方法来选择个体。根据适应值函数求解种群中全部个体的适应度。通过轮盘赌的算法选择个体。设置个体总数 M 个体 i 的适应值为 F_i，那么个体 i 被选中的概率为

$$P_i = \frac{F_i}{\sum_{i=1}^{M} F_i}, i = 1, 2, ...M$$

由公式显然可知，适应值高的个体被选中进入下一代的概率大，适应值低的个体被选中进入下一代的概率相对较少。

2. 锦标赛选择法

由于算法执行的效率以及易实现的的特点，锦标赛选择算法是遗传算法中最流行的选择策略。在本人的实际应用中的确此策略比基本的轮盘赌效果要好些。他的策略也很直观，就是我们再整个种群中抽取 n 个个体，让他们进行竞争(锦标赛)，抽取其中的最优的个体。参加锦标赛的个体个数成为锦标赛规模。通常当 $n = 2$ 便是最常使用的大小，也称作二进制锦标赛选择策略。

锦标赛选择法优势为：

1 更小的复杂度 $O(n)$

2 易并行化处理

3 不易陷入局部最优点

4 不需要对所有的适应度值进行排序处理

下图显示了 $n = 3$ 的锦标赛选择过程

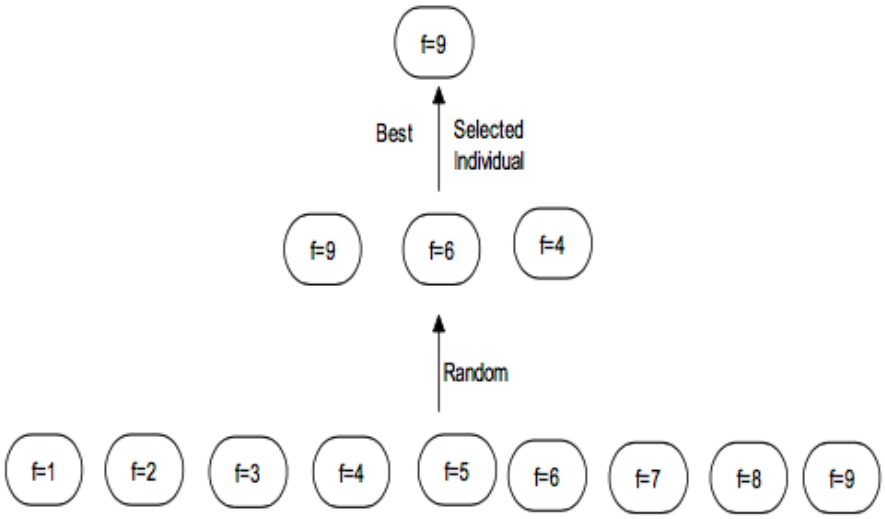


Fig. 3. Tournament Selection Mechanism [15]

3. 线性排序选择

下面两个介绍的选择策略都是基于排序的选择策略，上面提到的第一种基本轮盘赌选择算法，有一个缺点，就是如果一个个体的适应度值为 0 的话，则被选中的概率将会是 0，这个个体将不能产生后代。于是我们需要一种基于排序的算法，来给每个个体安排相应的选中概率。

在线性排序选择中，种群中的个体首先根据适应度的值进行排序，然后给所有个体赋予一个序号，最好的个体为 N，被选中的概率为 $P_{\max}$，最差的个体序号为 1，被选中的概率为 $P_{\min}$，于是其他的在他们中间的个体的概率便可以根据如下公式得到:

$$P_i = P_{\min} + \left(P_{\max} - P_{\min}\right)\frac{i-1}{N-1}$$

4. 指数排名选择

类似上面的 Linear Ranking 选择策略，这种指数排序便是在确定每个个体的选择概率的时候使用了指数形式的表达式，其中 c 为底数，满足 $0 < c < 1$:

$$P_i = \frac{c^{N-i}}{\sum_{j=1}^{N} c^{N-j}}$$

选择算子的比较与选择：

由于本模型中不产生适应度为 0 的数据所以暂时不考虑 3,4 种选择方法。

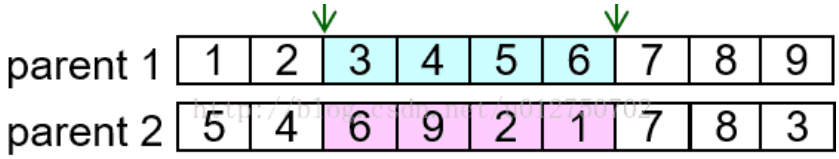
本模型分别采用了轮盘赌法和锦标赛选择法进行解模，并对得出的结果做了对比分析。

3.4 交叉操作

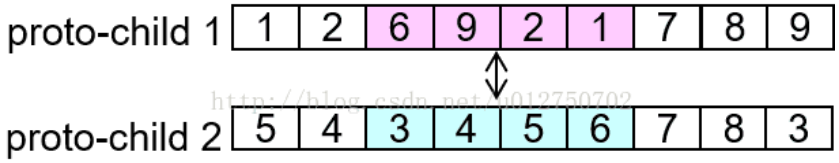
常见的交叉算子有：

1. Partial-Mapped Crossover (PMX)

第一步，随机选择一对染色体（父代）中几个基因的起止位置（两染色体被选位置相同）：

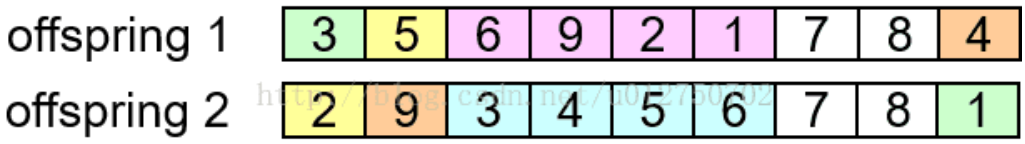


第二步，交换这两组基因的位置：



第三步，做冲突检测，根据交换的两组基因建立一个映射关系，如图所示，以 1-6-3 这一映射关系为例，可以看到第二步结果中子代 1 存在两个基因 1，这时将其通过映射关系转变为基因 3，以此类推至没有冲突为止。最后所有冲突的基因都会经过映射，保证形成的新一对子代基因无冲突：

最终结果

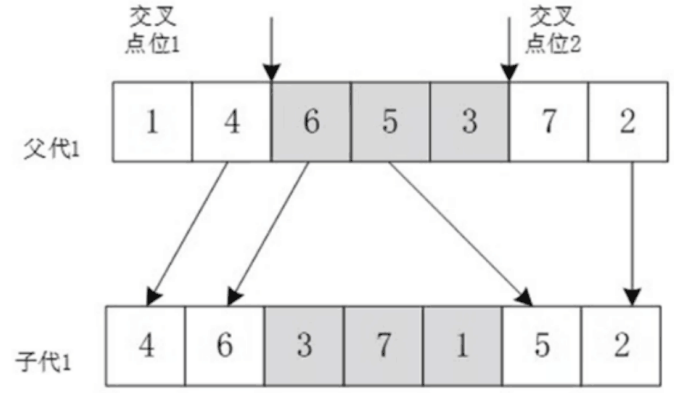


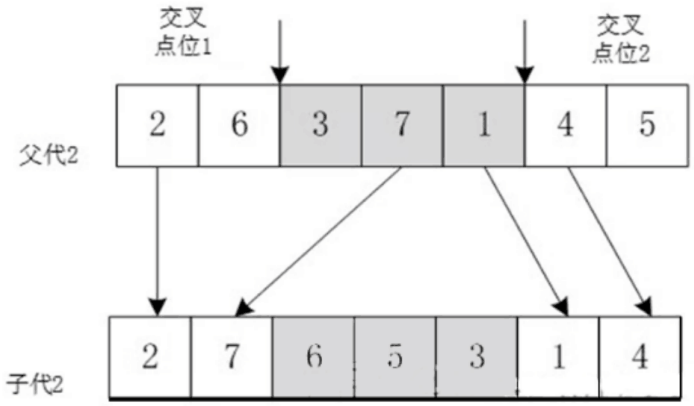
2. 顺序交叉

选择操作后，剩余被选择的个体可以进入交叉操作。种群的个体随机进行两两配对，假设配对的父代为父代 1 和父代 2。

第一步，父代 1 和父代 2 进行交叉操作。随机生成两个不同的基因点位作为几个基因的起止位置（两染色体被选位置相同）

第二步，生成一个子代，并保证子代中被选中的基因的位置与父代相同。子代 1 继承父代 2 交叉点位之间的基因片段，其余基因按顺序继承父代 2 中未重复的基因。如图所示：

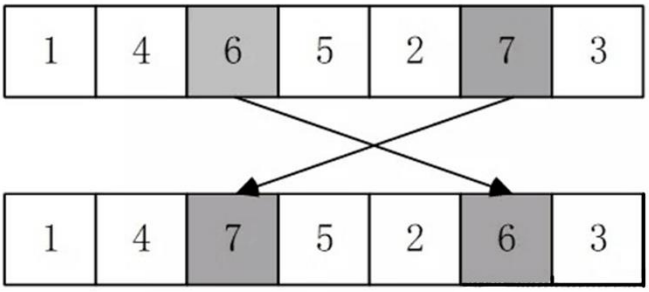




需要注意的是，这种算法同样会生成两个子代，另一个子代生成过程完全相同，只需要将两个父代染色体交换位置，第一步选中的基因型位置相同。
与 PMX 不同的是，不用进行冲突检测工作。
动车组检修作业问题是一类顺序排列问题，这类问题中，邻域信息有非常重要的作用，而位置信息影响不大。因此在设计杂交算子时，若考虑到保留较多的邻域信息，那么肯定有更好的性能，因此顺序交叉算子的性能比 PMX 交叉算子的性能好。
因此本模型选择了 OX 交叉算子。

3.5 变异操作

采用两点变异的方式，随机生成两个基因位，并交换两个基因位上的基因。



3.6 终止条件

常见的遗传算法终止条件有：

1、迭代固定次数后取最优值。

2、运行固定时间后取最优值。

3、确定一个优化目标，达到优化目标后就停止。

本文采取第一种终止条件，迭代固定次数后取最优值。

4. 算法测试

首先选取一组简单的测试数据对该算法进行测试：

工件数量 n=6

工序数量 c=3

每道工序并行机器数量 m=2

已知各工件各道工序的加工时间如下表所示：

	工序 1	工序 2	工序 3
工件 1	2	4	6
工件 2	4	9	2
工件 3	4	2	8
工件 4	9	5	6
工件 5	5	2	7
工件 6	9	4	3

遗传算法的关键参数，一般通过正交实验来定。本次测试取值如下：

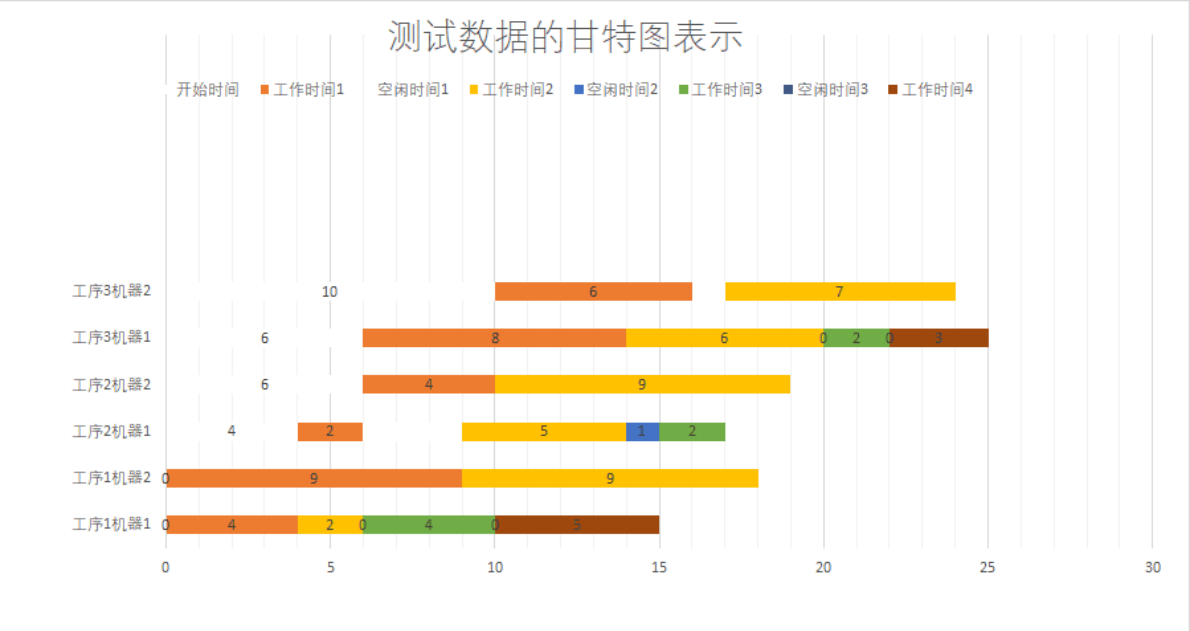
种群个数 N=200

交叉概率 0.6

变异概率 0.05

种群迭代次数 $G=100$

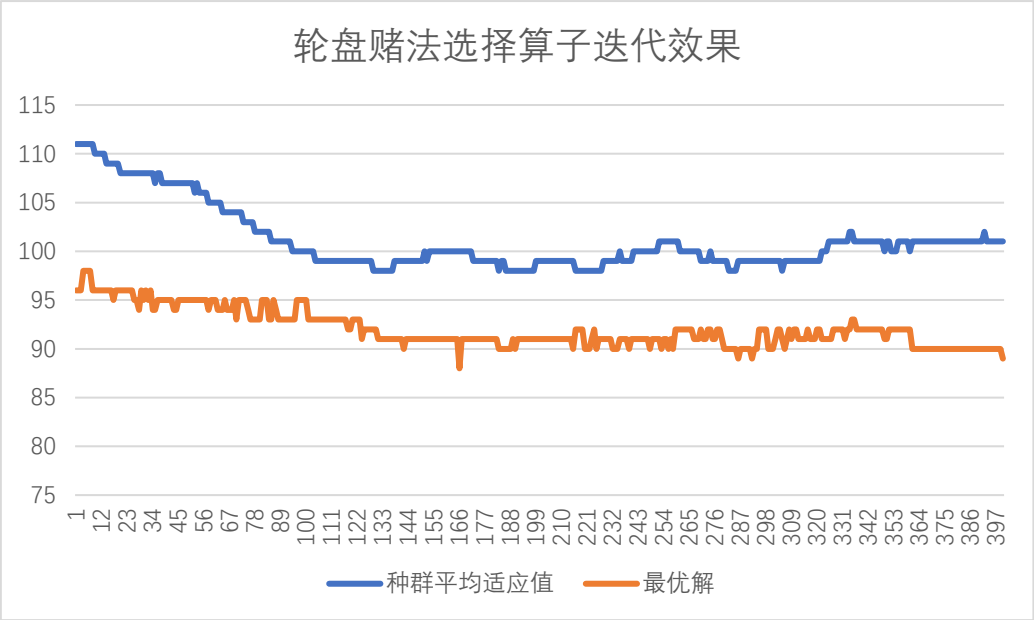
将运行结果用甘特图的形式表示出来，如下图所示：

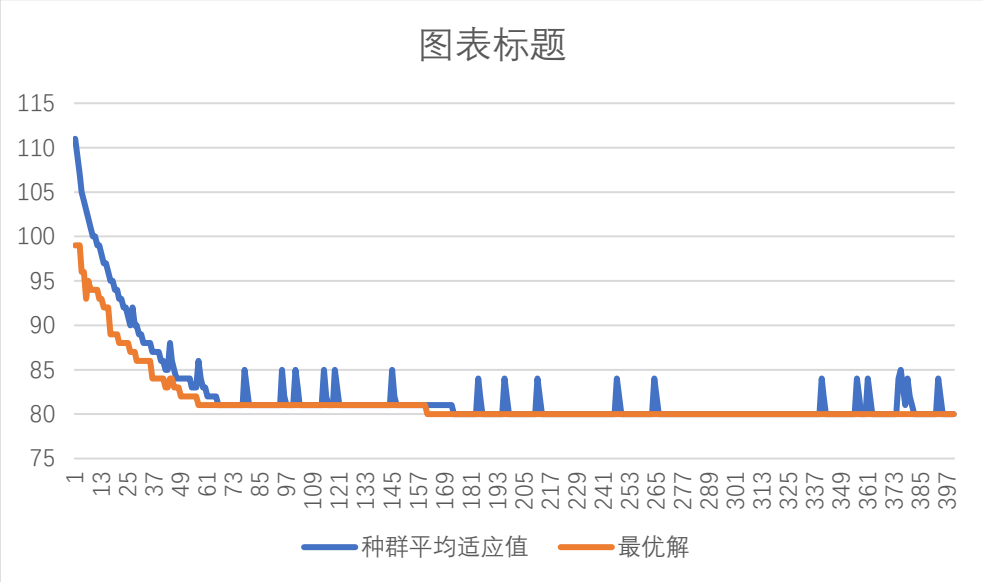


最优的加工顺序为 4，5，1，2，6，3，最终所用的时间为 25。验证得到的结果为全局最优解。

在 **3. 算法设计**的 **3. 3 选择操作**中，我们使用了两种不同的选择操作算子，分别为轮盘赌选择法和锦标赛选择法，为了对两种选择算法的优劣进行测试对比，我们选择了题目中问题一的数据作为测试样例。选择原因是题目一中动车数量较多，更考验选择算法的收敛速度和是否能收敛到全局最优解的性能。

种群数 $N=3000$，动车数量 $n=48$，交叉概率 0.6，变异概率 0.1，设置迭代次数为 400，输出每一次迭代的最优解和种群平均适应值，绘制折线图，使得收敛速度与收敛情况直观地呈现在折线图上。





由于终止条件设定为迭代固定次数后取最优值，经测试，锦标赛选择法在相同的种群数量和迭代次数的情况下，更容易收敛到全局最优解，且得到的最优解稳定不变，有收敛速度快和不容易陷入局部最优解的优点，因此我们选择锦标赛选择法作为本模型的选择算子。

5. 题目求解

根据论文 4.2 模型建立和 4.3 算法设计我们可以结合已经建立的遗传算法模型和该算法模型的具体设计过程解决下列三个问题。

下列三个问题共性都是求解目标函数——完工时间的最小值。并且目标函数受到动车检修调度顺序的约束。所以我们将遗传算法模型中的染色体设为动车检修优先级的随机全排列，进行求解。由于 4.2 和 4.3 部分已经对建立模型

和求解过程中的实现细节做了充分说明，此处便不再赘述。针对不同问题只修改模型的参数和初始条件进行求解。

问题一：

动车数量 $n=48$

工序数量 c=3

每道工序并行车间数量 m_1=3，m_2=8，m_3=5

每道工序耗费时间：$p(1,j)=4, p(2,j)=8, p(3,j)=6$ （单位：刻钟）

种群个数 N=500

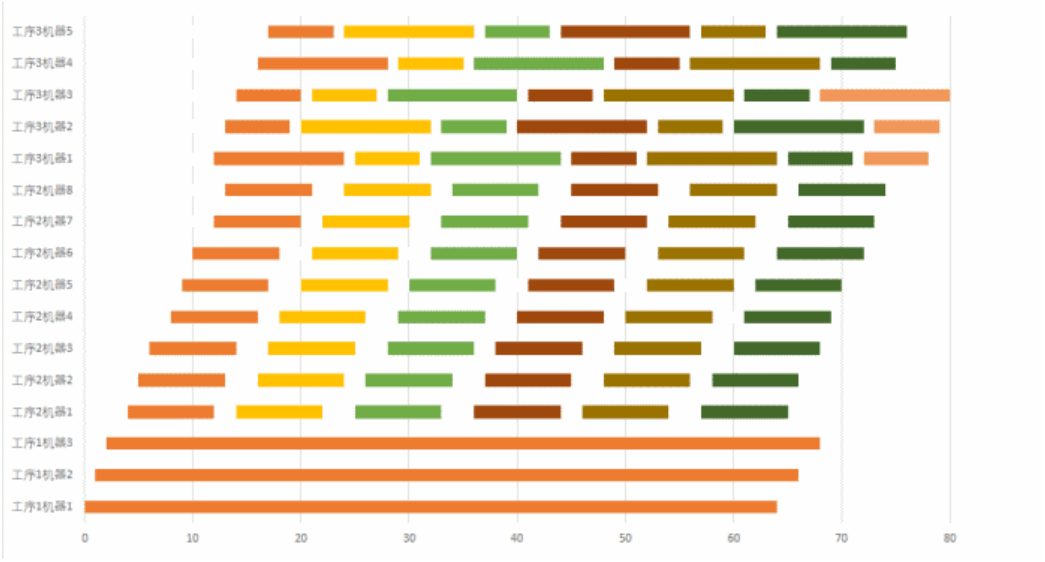
交叉概率 0.6

变异概率 0.02

种群迭代次数 $G=200$

将到站时差模拟成虚拟轮次 0 次的检修完成的时差。

将运行结果用甘特图的形式表示出来，如下图所示：
（图中横坐标的单位：刻钟）



最优加工顺序为 $1,2,3.....48$，按照列车到站顺序，最终用时为 20 小时。

结束时间：20:00

问题二：

动车数量$n=11$

工序数量$c=3$

每道工序并行车间数量$m_1=3$，$m_2=8$，$m_3=5$

已知各动车各道工序的加工时间如下表所示（单位：分钟）：

	工序 1	工序 2	工序 3
动车 1	60	120	90
动车 2	78	150	90
动车 3	60	120	90
动车 4	60	162	18
动车 5	48	144	30
动车 6	60	120	90
动车 7	60	162	18
动车 8	78	150	90
动车 9	48	144	30
动车 10	48	144	30
动车 11	60	162	18

到达时间表格（单位：分钟）

动车 1	16
动车 2	47
动车 3	82
动车 4	120
动车 5	141
动车 6	182
动车 7	211
动车 8	239
动车 9	241
动车 10	267
动车 11	309

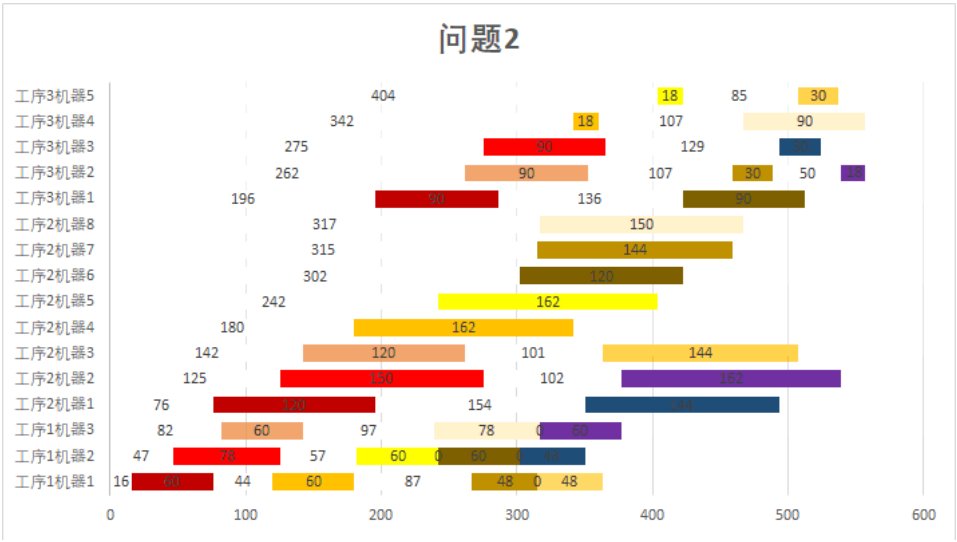
种群个数$N=500$

交叉概率0.6

变异概率0.02

种群迭代次数$G=200$

将运行结果用甘特图的形式表示出来，如下图所示：
（图中横坐标的单位：分钟）



最终用时为 557 分钟。**结束时间：9:17**

问题三：

动车数量 $n=7$

工序数量 c=5

每道工序并行车间数量 m_1=3，m_2=8，m_3=5

种群个数 N=2000

交叉概率 0.6

变异概率 0.02

种群迭代次数 $G=200$

已知各动车各道工序的加工时间如下表所示（单位：分钟）：

	a	b	c	d	e
动车 1	60	0	90	240	420
动车 2	78	150	90	0	0
动车 3	60	120	90	0	0
动车 4	60	162	0	0	0
动车 5	48	144	0	288	0
动车 6	60	162	18	0	0
动车 7	60	120	90	240	420

到达时间表格（单位：分钟）

动车 1	16
动车 2	47
动车 3	82

动车 4	120
动车 5	141
动车 6	182
动车 7	211

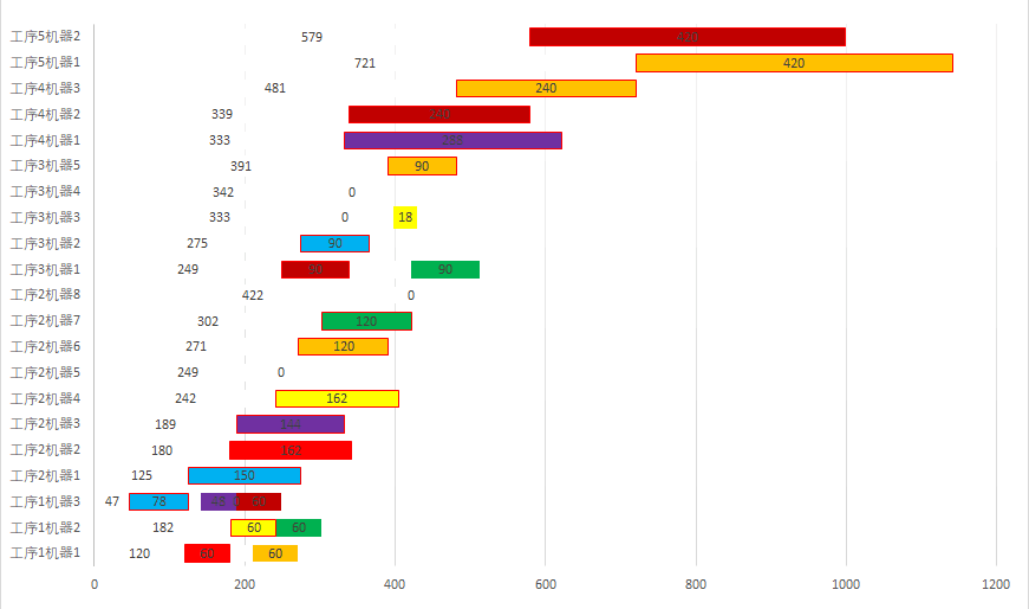
种群个数 $N=2000$

交叉概率 0.6

变异概率 0.02

种群迭代次数 $G=1000$

将运行结果用甘特图的形式表示出来，如下图所示：
（图中横坐标的单位：分钟）



最终用时为 1141 分钟。**结束时间：19:01**

五、模型评估

5.1 模型的优点：

1. 采用遗传算法作为核心模型，相较其他算法可以完成计算规模较大的时候的计算和寻优结果具备全局特性，克服局部最优等优点。

2. 交叉算子采用顺序交叉算子，相较 PMX 交叉算子保留了更多的邻域信息，有更好的模拟性能。

3. 算法架构有较好的兼容性，可以完成到达时间不同时，加工路径不相

同，不同车型在不同检修环节加工时间不一样等多种约束条件。

4.比较了两种选择算法：轮盘赌选择法和锦标赛选择法。设定固定迭代次数后，经测试，锦标赛选择法在相同的种群数量和迭代次数的情况下，更容易收敛到全局最优解，且得到的最优解稳定不变，有收敛速度快和不容易陷入局部最优解的优点，因此我们选择锦标赛选择法作为本模型的选择算子。

5.2 模型的不足：

1．由于题目中给出的约束条件和检修车组的数量规模并不是很大以及限于数据不足，所以在更大规模的车组检修作业求解，准确性和收敛速度仍需要进一步验证与改进。如果提供数据量充足，可以进一步测验更大规模数据下的模型性能，让模型更加可靠。

六、参考文献

[1]曹睿,侯向盼,金巳婷.基于改进遗传算法的柔性车间调度问题的研究[J].计算机与数字工程,2019,47(02):285-288.
[2] 王圣尧,王凌,许烨,周刚.求解混合流水车间调度问题的分布估计算法.自动化学报,2012,38(3):437-443.
[3]宋存利. 求解混合流水车间调度的改进贪婪遗传算法. 系统工程与电子技术 ,Systems Engineering and Electronics,2019,19(3)
[4]任海娇. 改进遗传算法求解流水车间调度问题[D]. 大连交通大学 2018
[5] tigerqin1980. 遗传算法求解混合流水车间调度问题. https://zhuanlan.zhihu.com/p/47921580 2018-10-29

使用锦标赛选择方法解决的第三问解题代码:

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <cstring>
#include <time.h>
#include <algorithm>
using namespace std;

ofstream outfile;
#define MAXPARALLEL 30
#define ordernumber 3     //工序数
#define workpiecesnumber 48  //工件总数
#define populationnumber 4000  //每一代种群的个体数
int parallel[ordernumber];

double crossoverrate = 0.6;           //交叉概率
double mutationrate = 0.1;            //变异概率
int G = 500;                        //循环代数 100
int usetime[workpiecesnumber][ordernumber];  //第几个工件第几道工序的加工用时;
int machinetime[ordernumber][MAXPARALLEL] = { 0 }; //第几道工序的第几台并行机器的统计时间;
int  starttime[workpiecesnumber][ordernumber][MAXPARALLEL];//第几个工件第几道工序在第几台并行机上开始加工的时间;
int finishtime[workpiecesnumber][ordernumber][MAXPARALLEL];//第几个工件第几道工序在第几台并行机上完成加工的时间;
int ttime[populationnumber];      //个体的 makespan;
int a[populationnumber][workpiecesnumber];//第几代的染色体顺序,即工件加工顺序;
int times[100];  //用来存储已知用时的数组;
int makespan;    //总的流程加工时间;
int flg7;   //暂时存储流程加工时间;
double fits[populationnumber];//存储每一代种群每一个个体的适应度,便于进行选择操作;

int tmpStore[populationnumber];//用来输出每代个体数值

int initialization()   //初始化种群;种群中个体相同 排列各不同
{
    for (int i = 0; i < populationnumber; i++)     //首先生成一个工件个数的全排列的个体;
        for (int j = 0; j < workpiecesnumber; j++)
        {
            a[i][j] = j + 1;
        }
```

```
    for (int i = 0; i < populationnumber; i++)      //将全排列的个体中随机选取
两个基因位交换，重复工件个数次，以形成随机初始种群;
        for (int j = 0; j < workpiecesnumber; j++)
        {
            int flg1 = rand() % workpiecesnumber;
            int flg2 = rand() % workpiecesnumber;
            int flg3 = a[i][flg1];
            a[i][flg1] = a[i][flg2];
            a[i][flg2] = flg3;
        }
    return 0;
}

void fitness(int c)   //计算适应度函数，c 代表某个体;
{
    int totaltime;      //总的加工流程时间（makespan);
    int temp1[workpiecesnumber] = { 0 };
    int temp2[workpiecesnumber] = { 0 };
    int temp3[workpiecesnumber] = { 0 };

    for (int j = 0; j < workpiecesnumber; j++)   //temp1 暂时存储个体 c 的基因
序列，以便进行不同流程之间的加工时记录工件加工先后顺序;
    {
        temp1[j] = a[c][j];
        temp2[j] = (a[c][j] - 1);
        temp3[j] = (a[c][j] - 1);
    }

    for (int i = 0; i < ordernumber; i++)
    {
        for (int j = 0; j < workpiecesnumber; j++)  //该循环的目的是通过比较所
有机器的当前工作时间，找出最先空闲的机器，便于新的工件生产;
        {
            int m = machinetime[i][0];         //先记录第 i 道工序的第一台并行机
器的当前工作时间;
            int n = 0;
            for (int p = 0; p < parallel[i]; p++) //与其他并行机器进行比较，找
出时间最小的机器;
            {
                if (m > machinetime[i][p])
                {
                    m = machinetime[i][p];
                    n = p;
```

```
                }
            }
            int q = temp1[j];                //按顺序提取 temp1 中的工件号，对工
件进行加工;
            starttime[q - 1][i][n] = max(machinetime[i][n], temp3[j]);  //开
始加工时间取该机器的当前时间和该工件上一道工序完工时间的最大值;
            machinetime[i][n] = starttime[q - 1][i][n] + usetime[q - 1][i];
//机器的累计加工时间等于机器开始加工的时刻，加上该工件加工所用的时间;
            finishtime[q - 1][i][n] = machinetime[i][n];                //工
件的完工时间就是该机器当前的累计加工时间;
            temp2[j] = finishtime[q - 1][i][n];       //将每个工件的完工时间赋
予 temp2，根据完工时间的快慢，便于决定下一道工序的工件加工顺序;
        }
        int flg2[workpiecesnumber] = { 0 };            //生成暂时数组，便于将
temp1 和 temp2 中的工件重新排列;
        for (int s = 0; s < workpiecesnumber; s++)
        {
            flg2[s] = temp1[s];
        }
        //
        for (int e = 0; e < workpiecesnumber - 1; e++)
        {
            for (int ee = 0; ee < workpiecesnumber - 1 - e; ee++) // 由于
temp2 存储工件上一道工序的完工时间，在进行下一道工序生产时，按照先完工先生产的
            {                                                  //原则，因此，该循环的目
的在于将 temp2 中按照加工时间从小到大排列，同时 temp1 相应进行变换
                if (temp2[ee] > temp2[ee + 1])               //来记录 temp2 中
的工件号;
                {
                    int flg5 = temp2[ee];
                    int flg6 = flg2[ee];
                    temp2[ee] = temp2[ee + 1];
                    flg2[ee] = flg2[ee + 1];
                    temp2[ee + 1] = flg5;
                    flg2[ee + 1] = flg6;
                }
            }
        }
        //
        for (int e = 0; e < workpiecesnumber; e++)    //更新 temp1，temp2 的数
据，开始下一道工序;
        {
            temp1[e] = flg2[e];
            temp3[e] = temp2[e];
```

```
        }
    }
    totaltime = 0;
    for (int i = 0; i < parallel[ordernumber - 1]; i++) //比较最后一道工序机器的累计加工时间，最大时间就是该流程的加工时间;
        if (totaltime < machinetime[ordernumber - 1][i])
        {
            totaltime = machinetime[ordernumber - 1][i];
        }
    for (int i = 0; i < workpiecesnumber; i++)  //将数组归零，便于下一个个体的加工时间统计;
        for (int j = 0; j < ordernumber; j++)
            for (int t = 0; t < parallel[j]; t++)
            {
                starttime[i][j][t] = 0;
                finishtime[i][j][t] = 0;
                machinetime[j][t] = 0;
            }
    makespan = totaltime;
    fits[c] = 1.000 / makespan;          //将 makespan 取倒数作为适应度函数;
}
void gant(int c)                    //该函数是为了将最后的结果便于清晰明朗的展示并做成甘特图，对问题的结果以及问题的解决并没有影响;
{
    int totaltime;
    char machine[ordernumber*MAXPARALLEL][500] = { "0" };
    int temp1[workpiecesnumber] = { 0 }; //加工顺序
    int temp2[workpiecesnumber] = { 0 }; //上一步骤的完成时间
    int temp3[workpiecesnumber] = { 0 };
    ////////////////////////////////////////
    for (int j = 0; j < workpiecesnumber; j++)
    {
        temp1[j] = a[c][j];
        temp2[j] = (a[c][j] - 1);
        temp3[j] = (a[c][j] - 1);
        cout << "******" << temp1[j] << endl;
    }
    for (int i = 0; i < ordernumber; i++)
    {
        for (int j = 0; j < workpiecesnumber; j++)
        {
            int m = machinetime[i][0];
            int n = 0;
            for (int p = 0; p < parallel[i]; p++) //找出时间最小的机器;
```

```
            {
                if (m > machinetime[i][p])
                {
                    m = machinetime[i][p];
                    n = p;
                }
            }
            int q = temp1[j];
            starttime[q - 1][i][n] = max(machinetime[i][n], temp3[j]);
            machinetime[i][n] = starttime[q - 1][i][n] + usetime[q - 1][i];
            finishtime[q - 1][i][n] = machinetime[i][n];
            temp2[j] = finishtime[q - 1][i][n];

            for (int h = starttime[q - 1][i][n]; h < finishtime[q - 1][i][n];
h++)
            {
                if (q >= 0 && q < 26)
                    machine[i*MAXPARALLEL + n][h] = 'a' - 1 + q;
                else
                    machine[i*MAXPARALLEL + n][h] = 'A' + (q - 26);
            }
        }
        int flg2[workpiecesnumber] = { 0 };
        for (int s = 0; s < workpiecesnumber; s++)
        {
            flg2[s] = temp1[s];
        }
        for (int e = 0; e < workpiecesnumber - 1; e++)
        {
            for (int ee = 0; ee < workpiecesnumber - 1 - e; ee++)
            {
                if (temp2[ee] > temp2[ee + 1])
                {
                    int flg5 = temp2[ee];
                    int flg6 = flg2[ee];
                    temp2[ee] = temp2[ee + 1];
                    flg2[ee] = flg2[ee + 1];
                    temp2[ee + 1] = flg5;
                    flg2[ee + 1] = flg6;
                    //swap(temp2[ee],temp2[ee+1]);
                    //swap(flg2[ee],flg2[ee+1]);
                }
            }
        }
```

```
        for (int e = 0; e < workpiecesnumber; e++)
        {
            temp1[e] = flg2[e];
            temp3[e] = temp2[e];
            //cout<<"temp3=="<<temp3[e]<<endl;
        }
    }
    totaltime = 0;
    for (int i = 0; i < parallel[ordernumber - 1]; i++)
        if (totaltime < machinetime[ordernumber - 1][i])
        {
            totaltime = machinetime[ordernumber - 1][i];
        }
    cout << "total=" << totaltime << endl;
    outfile << totaltime << endl;
    flg7 = totaltime;

    int idx = 0;
    for (int i = 0; i < ordernumber; i++)
    {
        for (int u = 0; u < parallel[i]; u++)
        {
            for (int uu = 0; uu < 100; uu++)
            {
                outfile << machine[idx + u][uu];
                cout << machine[idx + u][uu];
            }
            outfile << endl;
            cout << endl;
        }
        idx += MAXPARALLEL;
    }
}
void select()
{
    int tmp[populationnumber][workpiecesnumber] = { 0 };
    int tmpfits[populationnumber] = { 0 };
    for (int i = 0; i < populationnumber; i++)
    {
        int flg1 = rand() % populationnumber;
        int flg2 = rand() % populationnumber;
        if (fits[flg1] > fits[flg2])
        {
            for (int m = 0; m < workpiecesnumber; m++)
```

```
            {
                tmp[i][m] = a[flg1][m];
            }
            tmpfits[i] = fits[flg1];
        }
        else
        {
            for (int m = 0; m < workpiecesnumber; m++)
            {
                tmp[i][m] = a[flg2][m];
            }
            tmpfits[i] = fits[flg2];
        }
    }
    for (int i = 0; i < populationnumber; i++)
    {
        for (int m = 0; m < workpiecesnumber; m++)
        {
            a[i][m] = tmp[i][m];
        }
        fits[i] = tmpfits[i];
    }
}
void crossover()
/*种群中的个体随机进行两两配对，配对成功的两个个体作为父代 1 和父代 2 进行交叉操作。
随机生成两个不同的基因点位，子代 1 继承父代 2 基因位之间的基因片段，其余基因按顺序集
成父代 1 中未重复的基因；
子代 2 继承父代 1 基因位之间的基因片段，其余基因按顺序集成父代 2 中未重复的基因。*/
{
    for (int i = 0; i < populationnumber / 2; i++) //将所有个体平均分成两部分，
一部分为交叉的父代 1，一部分为进行交叉的父代 2；
    {
        int n1 = 1 + rand() % workpiecesnumber / 2;    //该方法生成两个不同的
基因位；
        int n2 = n1 + rand() % (workpiecesnumber - n1 - 1) + 1;
        int n3 = rand() % 10;
        if (n3 == 2)   //n3=2 的概率为 0.1；若满足 0.1 的概率，那么就进行交叉操作；
        {
            int temp1[workpiecesnumber] = { 0 }; int temp2[workpiecesnumber]
= { 0 };
            for (int j = 0; j < workpiecesnumber; j++)
            {
                int flg1 = 0; int flg2 = 0;
                for (int p = n1; p < n2; p++)          //将交叉点位之间的基因片
```

段进行交叉，temp1 和 temp2 记录没有发生重复的基因；

```
                {
                    if (a[2 * i + 1][p] == a[2 * i][j])
                        flg1 = 1;
                }
                if (flg1 == 0) { temp1[j] = a[2 * i][j]; }

                for (int p = n1; p < n2; p++)
                {
                    if (a[2 * i][p] == a[2 * i + 1][j])
                        flg2 = 1;
                }
                if (flg2 == 0) { temp2[j] = a[2 * i + 1][j]; }

            }


            for (int j = n1; j < n2; j++)             //子代 1 继承父代 2 交叉点
```

位之间的基因；子代 2 继承父代 1 交叉点位之间的基因；

```
            {
                int n4 = 0;
                n4 = a[2 * i][j];
                a[2 * i][j] = a[2 * i + 1][j];
                a[2 * i + 1][j] = n4;
            }
            for (int p = 0; p < n1; p++)              //子代 1 第一交叉点之前的
```

基因片段，按顺序依次继承父代 1 中未与子代 1 重复的基因；

```
            {
                for (int q = 0; q < workpiecesnumber; q++)
                {
                    if (temp1[q] != 0)
                    {
                        a[2 * i][p] = temp1[q]; temp1[q] = 0;
                        break;
                    }
                }
            }
            for (int p = 0; p < n1; p++)              //子代 2 第一交叉点之前的
```

基因片段，按顺序依次继承父代 2 中未与子代 2 重复的基因；

```
            {
                for (int m = 0; m < workpiecesnumber; m++)
                {
                    if (temp2[m] != 0)
                    {
```

```
                        a[2 * i + 1][p] = temp2[m]; temp2[m] = 0;
                        break;
                    }
                }
            }
            for (int p = n2; p < workpiecesnumber; p++)            //子代 1 第
2 交叉点之后的基因片段，按顺序依次继承父代 1 中未与子代 1 重复的基因;
            {
                for (int q = 0; q < workpiecesnumber; q++)
                {
                    if (temp1[q] != 0)
                    {
                        a[2 * i][p] = temp1[q]; temp1[q] = 0;
                        break;
                    }

                }
            }
            for (int p = n2; p < workpiecesnumber; p++)              //子代 2
第 2 交叉点之后的基因片段，按顺序依次继承父代 2 中未与子代 2 重复的基因;
            {
                for (int m = 0; m < workpiecesnumber; m++)
                {
                    if (temp2[m] != 0)
                    {
                        a[2 * i + 1][p] = temp2[m]; temp2[m] = 0;
                        break;
                    }
                }
            }

        }
    }


}

////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////

void mutation()  //变异操作为两点变异，随机生成两个基因位，并交换两个基因的位置;
{
    int n3 = rand() % 20;
    if (n3 == 2)
```

```
    {
        for (int i = 0; i < populationnumber; i++)
        {
            int b1 = rand() % workpiecesnumber;
            int b2 = rand() % workpiecesnumber;
            int b3 = a[i][b1];
            a[i][b1] = a[i][b2];
            a[i][b2] = b3;
        }
    }
}
int main()
{
    int sum = 0; int min = 10000;
    for (int i = 0; i < workpiecesnumber; i++)
    {
        usetime[i][0] = 4; usetime[i][1] = 8; usetime[i][2] = 6;
    }
    parallel[0] = 3; parallel[1] = 8; parallel[2] = 5;
    cout << "////////////////////////////////////////////////" << endl;
    srand(time(NULL));
    initialization();    //初始化种群;
    for (int g = 0; g < G; g++)
    {
        sum = 0; min = 10000;
        for (int c = 0; c < populationnumber; c++)//计算每个个体适应度并存在
ttime 中;
        {
            fitness(c);
            ttime[c] = makespan;
        }
        cout << "*******************************" << endl;
        for (int i = 0; i < populationnumber; i++)
            tmpStore[i] = ttime[i];
        sort(tmpStore, tmpStore + populationnumber);
        for (int i = 0; i < populationnumber; i++)
        {
            sum += tmpStore[i];
            if (min > tmpStore[i]) min = tmpStore[i];
        }
        cout << "平均值: " << sum / populationnumber << "最优解: " << min <<
endl;
        cout << "*******************************" << endl;
        select();     //选择操作;
```

```
        crossover();  //交叉操作;
        mutation();   //变异操作;
    }
    for (int c = 0; c < populationnumber; c++)//计算每个个体适应度并存在 ttime
中;
    {
        fitness(c);
        ttime[c] = makespan;
    }
    int flg8 = ttime[0];
    int flg9 = 0;
    for (int c = 0; c < populationnumber; c++)  //计算最后一代每个个体的适应度,
并找出最优个体;
    {
        if (ttime[c] < flg8)
        {
            flg8 = ttime[c];
            flg9 = c;
        }
    }
    gant(flg9);   //画出简易的流程图;
    outfile.close();
    system("pause");
    return 0;
}
```

轮盘赌选择法选择代码:

```
void select()
{
    double roulette[populationnumber + 1] = { 0.00 };    //记录轮盘赌的每一个概率区间;
    double pro_single[populationnumber];            //记录每个个体出现的概率，即个体的
适应度除以总体适应度之和;
    double totalfitness = 0.00;                 //种群所有个体的适应度之和;
    int a1[populationnumber][workpiecesnumber]; //存储a中所有个体的染色体;

    for (int i = 0; i < populationnumber; i++)          //计算所有个体适应度的总和;
    {
        totalfitness = totalfitness + fits[i];
    }
    for (int i = 0; i < populationnumber; i++)
    {
        pro_single[i] = fits[i] / totalfitness;     //计算每个个体适应度与总体适应度之
比;
        roulette[i + 1] = roulette[i] + pro_single[i]; //将每个个体的概率累加，构造轮
```

```
盘赌；
    }
    for (int i = 0; i < populationnumber; i++)
    {
        for (int j = 0; j < workpiecesnumber; j++)
        {
            a1[i][j] = a[i][j];                //a1暂时存储a的值；
        }
    }
    for (int i = 0; i < populationnumber; i++)
    {
        int a2;   //当识别出所属区间之后，a2记录区间的序号；
        double p = rand() % (1000) / (double)(1000);
        for (int j = 0; j < populationnumber; j++)
        {
            if (p >= roulette[j] && p < roulette[j + 1])
                a2 = j;
        }
        for (int m = 0; m < workpiecesnumber; m++)
        {
            a[i][m] = a1[a2][m];
        }
    }
}
```