# 摘　要

本文通过使用遗传算法对作业车间调度问题进行建模和研究，并用 C++ 程序实现了整个遗传算法的操作过程，得到了较好的结果。

首先针对作业调度问题各工件的工序及加工所使用的机器都有所不同，为了尽可能简单地用一个数字序列作为染色体基因序列来表示工序的调度，本文采取基于工序的表示法来构成染色体，通过简单的映射得到工序的详细信息，之后便可以计算任何一个可调度的工序集合的最大完工时间，并采用其倒数作为适应度函数的值，衡量各染色体的适应程度。

针对遗传算法的三个步骤：选择操作、交叉操作和变异操作，本文采取了多种策略相结合的方式进行操作。对于选择操作，先使用精英保留策略选出一部分个体，再通过轮盘赌策略选择出剩余所需要的个体；对于交叉操作，本文采用顺序交叉和单点交叉相结合的方法进行交叉来保证一定可以产生可行调度，避免了之后需要进行修正的问题，而且根据适应度函数的变化改变种群的交叉概率，可以尽可能地保留优秀的个体；对于变异操作，本文使用选择染色体子段进行对调的方法进行操作，同时也根据个体适应度与适应度平均值的关系来确定变异概率，一定程度上避免了陷入局部最优解的问题。

最后通过多次模拟实验进行仿真，发现在迭代 30 次时基本已经稳定到最优解，而且通过增大初始种群的规模，有效地提高了达到最优解的稳定性，在初始种群规模为 40 时，已经可以得到很稳定的结果。



**关键字：遗传算法　作业车间调度**

## 1 问题重述

墙纸生产商需要生产三种类型的墙纸[1]，而且每种墙纸都要印刷若干种不同颜色的图案，每种墙纸的印刷顺序[2]各不相同，印刷每种颜色所用的机器以及耗时均已给出，要求制定印刷计划，使印刷尽快完成。

## 2 条件假设

1. 每个工件的工序不是随意安排的，存在加工顺序约束。
2. 一台机器同一时刻只能加工一个工件的某一个工序。
3. 机器不存在故障可以连续运转，每一个墙纸的某一工序开始加工就不能停止。
4. 工件的每道工序在指定机器上加工的时间是确定并且已知的。
5. 每个工件利用每台机器的顺序可以不同。
6. 任何工件都没有抢先加工的优先权，服从指定的加工顺序
7. 工件加工过程中没有新工件的加入，也不临时取消工件的加工
8. 加工相同工序的机器有且只有一台

## 3 符号说明

$n$：工件的个数

$p_i$：第 $i$ 个工件，$i = 1, 2, \ldots, n$

$sn_i$：第 $i$ 号机器的工序个数，$i = 1, 2, \ldots, n$

$op_{ij}$：第 $i$ 号机器的第 $j$ 道工序，$i = 1, 2, \ldots, n; j = 1, 2, \ldots, sn_i$

$m_{ij}$：第 $i$ 个工件的第 $j$ 道工序所使用的机器编号，$i = 1, 2, \ldots, n; j = 1, 2, \ldots, sn_i$

$t_{ij}$：完成第 $i$ 个工件的第 $j$ 道工序需要花费的时间，$i = 1, 2, \ldots, n; j = 1, 2, \ldots, sn_i$

$N$：种群中染色体的条数

$L$：每条染色体的基因个数

$s_{ij}$：第 $i$ 个种群的第 $j$ 个基因所代表的工序，$i = 1, 2, \ldots, N; j = 1, 2, \ldots, L$

$ct_{ij}$：按照染色体的基因序列，第 $i$ 个工件完成前 $j$ 道工序所花费的总时间，$i = 1, 2, \ldots, n; j = 1, 2, \ldots, sn_i$

$mt_{ij}$：按照染色体的基因序列，到第 $i$ 个工件完成前 $j$ 道工序为止，机器 $m_{ij}$ 所花费的总时间，$i = 1, 2, \ldots, n; j = 1, 2, \ldots, sn_i$

$T_i$：种群中第 $i$ 个染色体完成所有工序所花费的时间，$i = 1, 2, \ldots, N$

$f_i$：种群中第 $i$ 个染色体的适应性函数，$i = 1, 2, \ldots, N$

[1]正文中用“工件”一词代替

[2]正文中用“工序”一词代替

表 1: 染色体解释表

| 基因 | 2 | 3 | 2 | 3 |
|---|---|---|---|---|
| 工序 | (2,1,1,10) | (3,1,3,28) | (2,2,2,20) | (3,2,2,12) |
| 基因 | 1 | 1 | 3 | 2 |
| 工序 | (1,1,2,45) | (1,2,3,10) | (3,3,1,17) | (2,3,3,34) |

# 4 问题分析及模型建立

## 4.1 基因编码

### 4.1.1 编码方式的选择

在作业车间调度问题中，由于工件的顺序，以及每个工序所使用的机器、耗费的时间都有所不同，对于 $n$ 个工件的调度问题，我们采用基于工序的表达法进行编码。

由基于工序的表达法得到的染色体由一组数字组成，一个数字代表一个工件的某一道工序，而出现在染色体不同位置上的同一个数字则代表该工件的不同工序，其中第 $j$ 次出现的数字 $i$ 即代表 $op_{ij}$。

因此所有的基因所表示的意义都不是独立存在的，而是与染色体上其它的基因有着上下文的依赖关系，也正是这种依赖关系的存在节省了空间，使得单条染色体的长度尽可能短，对于 $n$ 个工件的调度问题来说，单条染色体的长度为 $\sum_{i=1}^{n} sn_i$。

### 4.1.2 基因与工序的映射关系

我们建立这样的一个四元组来表示每一道工序的加工细节，$(i, j, k, t)$ 表示第 $i$ 个工件的第 $j$ 道工序在第 $k$ 号机器上进行加工，而且加工的时间为 $t$。这样一来，对于染色体中的每一条基因，都可以唯一地映射到这样地一个四元组上，从而产生可行的调度。

对于题目中给定的数据，我们指定绿色机器为 1 号机器，蓝色机器为 2 号机器，黄色机器为 3 号机器。对于第 $i$ 个工件，都可以得到 $sn_i$ 个四元组，分别为:

工件 1：$(1,1,2,45)$ $(1,2,3,10)$

工件 2：$(2,1,1,10)$ $(2,2,2,20)$ $(2,3,3,34)$

工件 3：$(3,1,3,28)$ $(3,2,2,12)$ $(3,3,1,17)$

我们随机产生一个染色体序列：2 3 2 3 1 1 3 2

它对应的可行调度如表 1所示。

### 4.1.3 程序实现

本文程序中定义了函数 $produceChrom()$ 来随机产生一个符合要求可以产生可行调度的染色体，对于初始化一个大小为 $N$ 的种群，循环调用该函数 $N$ 次即可。

表 2: $ct_{ij}$ 数值表

| 工序 / 工件 | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 87 | 97 | |
| 2 | 10 | 30 | 131 |
| 3 | 28 | 42 | 59 |

## 4.2 基因解码及适应性函数计算

### 4.2.1 基因解码及求解目标函数

依据问题的描述，需要达到的目标为最小化最大完工时间，对于种群中的第 $i$ 个染色体而言，最大完工时间即为：

$$T_i = max_{1 \le i \le n; j = sn_i} ct_i j \tag{1}$$

我们希望可以通过遗传迭代，使得这一时间尽可能地小。

染色体的解码过程分为两个步骤：首先将染色体翻译为工序序列，如表 1，然后依据得到的工序序列安排每道工序到指定的机器上运行，在安排任务的同时要考虑到如下的限制约束：

1. 每台机器同一时间只能加工一个工件的一道工序。

2. 每个工件的工序都是确定的，不可发生改变。

由这两个约束，我们可以写出如下表达式：

$$ct_{ij} = max(ct_{i\ j-1}, mt_{ij}) + t_{ij} \tag{2}$$

即每个工序完成加工的时间都取决于该工件上一工序完成的时间、机器完成上一工序的时间以及该工序所花费的时间

$$mt_{ij} = ct_{ij} \tag{3}$$

即某工序加工完成后，加工该工序的机器所花费的时间应该与该工序加工完成的时间相同。

结合式 1,2,3以及给定的基因序列，便可以求出针对该染色体所对应调度的最大完工时间。

### 4.2.2 求解示例及分析

我们仍然使用之前的随机序列：2 3 2 3 1 1 3 2
根据公式我们可以做出该序列调度的 *Gantt* 图 1和 $ct_{ij}$ 的表格 2。

很容易发现，如果仅根据约束条件来看，应该得到图 2所示的结果，但是通过程序计算出的完工时间为 131，从而得到图 1的结果，这是因为在工件 1 的第二个工序执行完后，机器的用时已经被置为 97，因此图中 3 号机器执行工件 2 的第三个工序是在工件 1 的第二个工序之后的。

但是由于工件 1 的第二道工序先执行，如果考虑充分利用空隙时间，将大大增加算法的复杂程度，然而只要将这道工序移至最后，表现在染色体中即为将最后的 1 和 2 对调，就可以得到图 2的结果，因此我们通过遗传迭代产生新染色体的过程来解决这个问题。

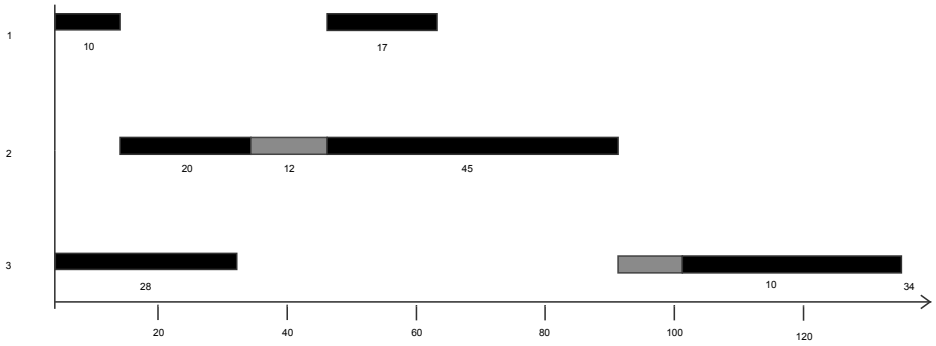


图 1: 流程 *Gantt* 图 1

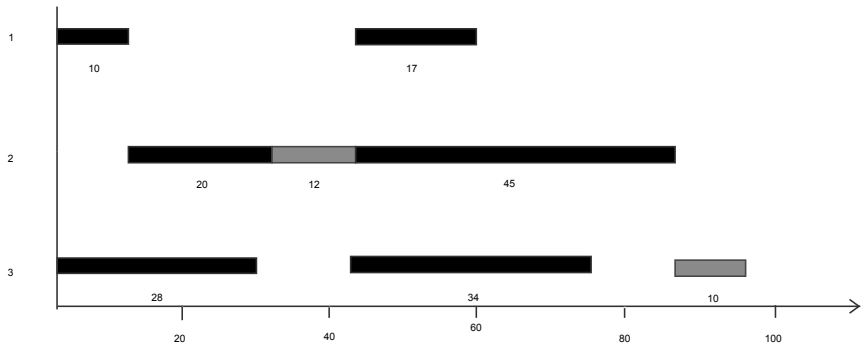


图 2: 流程 *Gantt* 图 2

#### 4.2.3 适应性函数的计算

因为适应函数越大的个体，应该有更大的生存概率，而作为目标的完工时间则是越小越好，另外考虑到适应函数应该易于计算、单值且非负的特点，我们选取

$$f_i = \frac{1}{T_i}, i = 1, 2, \ldots, N \tag{4}$$

作为种群中第 $i$ 个染色体的适应性函数。

#### 4.2.4 程序实现

本文程序中定义了函数 $cal\_time()$ 和 $cal\_suit()$ 来根据给定的染色体基因序列分别计算完工时间的值和适应度函数的值。

### 4.3 选择操作

本文中选择操作采用精英保留策略和轮盘赌策略相结合的方法来进行操作。

#### 4.3.1 精英保留策略

首先，我们要计算出当前种群中所有染色体的适应度函数的平均值。

$$f_{avg} = \alpha * \frac{\sum_{i=1}^{N} f_i}{N} \tag{5}$$

然后遍历所有的染色体，如果满足 $f_i > f_{avg}, i = 1, 2, \ldots, N$，则该染色体会生存下来参与下一次遗传迭代。

公式中引入了一个比例系数 $\alpha$, 是因为如果直接采用适应度的平均值作为比较基础，将很容易陷入局部最优解中，因此选用一个略大于 1 的比例系数 $\alpha$, 可以选出相对更加精英的染色体，在一定程度上避免了这种情况的发生。

#### 4.3.2 轮盘赌策略

在遗传进化初期，由于精英染色体个数较少，一般不会超过种群染色体总数的一半，因此还需要再选择一部分个体进入下一代，这里使用轮盘赌的策略来实现。

首先通过适应度函数计算出染色体可能被选中的概率。

$$p_i = \frac{f_i}{\sum_{i=1}^{N} f_i} \tag{6}$$

然后根据每一个 $p_i$ 值的大小，为第 $i$ 个染色体分配大小为 $p_i$ 的区间，之后随机产生一个 $[0, 1]$ 之间的随机数 $r$，$r$ 落到哪个区间范围内，则将该个体保留下来，直到新种群的大小达到需求大小为止。

#### 4.3.3 程序实现

本文中定义了函数 $selectChrom()$ 来实现选择操作，首先通过经营保留策略选择一部分染色体，再通过轮盘赌策略选择剩余的一部分染色体进入下一代。

### 4.4 交叉操作

对于基于工序表达的染色体，如果直接使用单点交叉操作，将产生无意义的染色体，不能映射为一个可行调度，需要进行修正，因此我们采用单点交叉和顺序交叉操作相结合的方式来进行操作。

#### 4.4.1 交叉基本操作

对于两个染色体 $S_1, S_2$，随机选取一个小于 $L$ 的数字 $r$，将两个染色体中前 $i$ 个基因提取出来作为子代的前 $r$ 个基因，分别记为 $R_1, R_2$。再根据 $R_2$ 中的基因序列遍历染色体 $S_1$，找到相同的基因即将其删除，然后将 $S_1$ 中剩余的基因接在 $R_2$ 后面，即产生了一条新的染色体；同理，对染色体 $S_2$ 进行相似的操作也会产生一条新的染色体。具体实例见图 3。

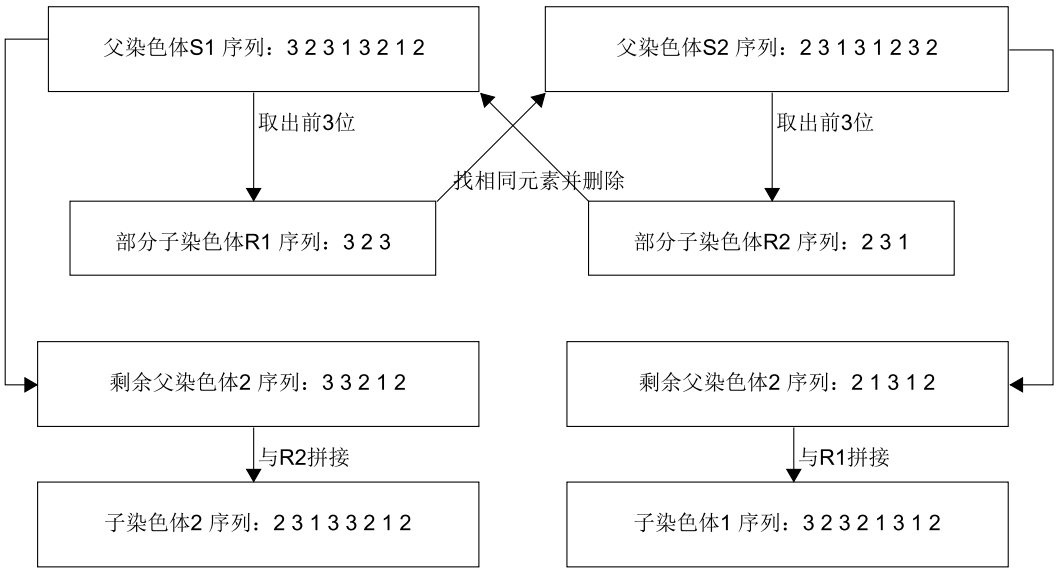


图 3: 交叉操作示例

#### 4.4.2 动态调整交叉概率

在遗传进化的后期，群体中染色体的适应度值不断靠近最优解，如果还是有比较大的变异概率，那么可能会破坏许多优秀基因，增大算法的迭代次数，因此我们对交叉规模进行动态调整，使其根据适应度函数的平均值的增大而减小。

交叉概率为：

$$p_c = p_{c0} - \frac{(f_{avg} - f'_{avg}) * p_{cst}}{f'_{avg}} \tag{7}$$

$p_{c0}$ 为交叉概率设置的初始值

$f_{avg}$ 为到当代数种群适应度值得平均值

$f'_{avg}$ 为初代种群的适应度平均值

$p_{cst}$ 为交叉概率成长值

### 4.4.3 程序实现

本文定义了函数 $crossOverChrom()$ 函数来实现交叉操作，交叉的次数为 $t = p_c * N$，交叉的序列是随机选取的两个小于等于 $N$ 的正整数代表的染色体。

## 4.5 变异操作

与交叉操作的局限性类似，如果简单地改变染色体某位置上的基因值，将产生无意义的调度序列，这里我们通过对调来方式无效调度序列的产生。

### 4.5.1 变异基本操作

在染色体 $S$ 上随机选取两个不同的点，将这两个点之间的所有基因提取出来，进行翻转，得到与之前子段完全相反的基因序列，再插入染色体 $S$ 中，即可产生一个新的染色体，并且对应一个可行的调度序列。具体实例见图 4。

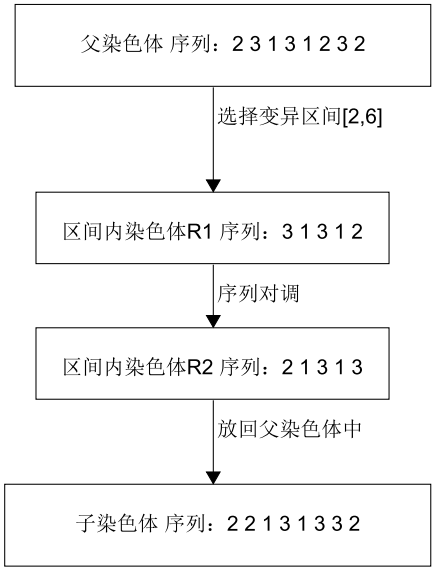


图 4: 变异操作示例

### 4.5.2 动态调整交叉概率

在进化后期，种群整体的平均适应度值和最优个体的适应度值非常接近，因此如果变异概率一直保持不变，就会使群体缺乏竞争性，优秀个体得不到保留，差的个体又不会被淘汰，从而导致陷入局部最优解的情况。因此我们在进化过程中动态改变变异发生的概率，使其根据个体的适应度做相应的调整。使得对于某一个体而言，它的适应度值越高，变异的概率就越小。变异概率为:

$$p_m = \begin{cases} p_{m0} - \frac{(f_i - f_{avg}) * p_{mst}}{f_{avg}} & f_i > f_{avg} \\ p_{m0} & f \leq f_{avg} \end{cases} \tag{8}$$

$p_{m0}$ 为变异概率设置的初始值
$f$ 为该个体的适应度函数值

$f_{avg}$ 为种群的适应度平均值
$p_{mst}$ 为变异概率成长值

### 4.5.3 程序实现

本文中定义了函数 $mutateChrom()$ 函数来实现这种对调的变异操作，通过每个染色体的 $p_m$ 值的大小确定发生变异的染色体，通过产生随机数确定变异的范围，之后通过对调进行变异产生新的染色体。

## 5 实例求解结果

程序执行的具体流程逻辑如图 5所示。
本文程序中定义函数 $manipulate()$ 来执行整个流程，包括判断终止条件，以及确定交叉操作和变异操作的相应概率。

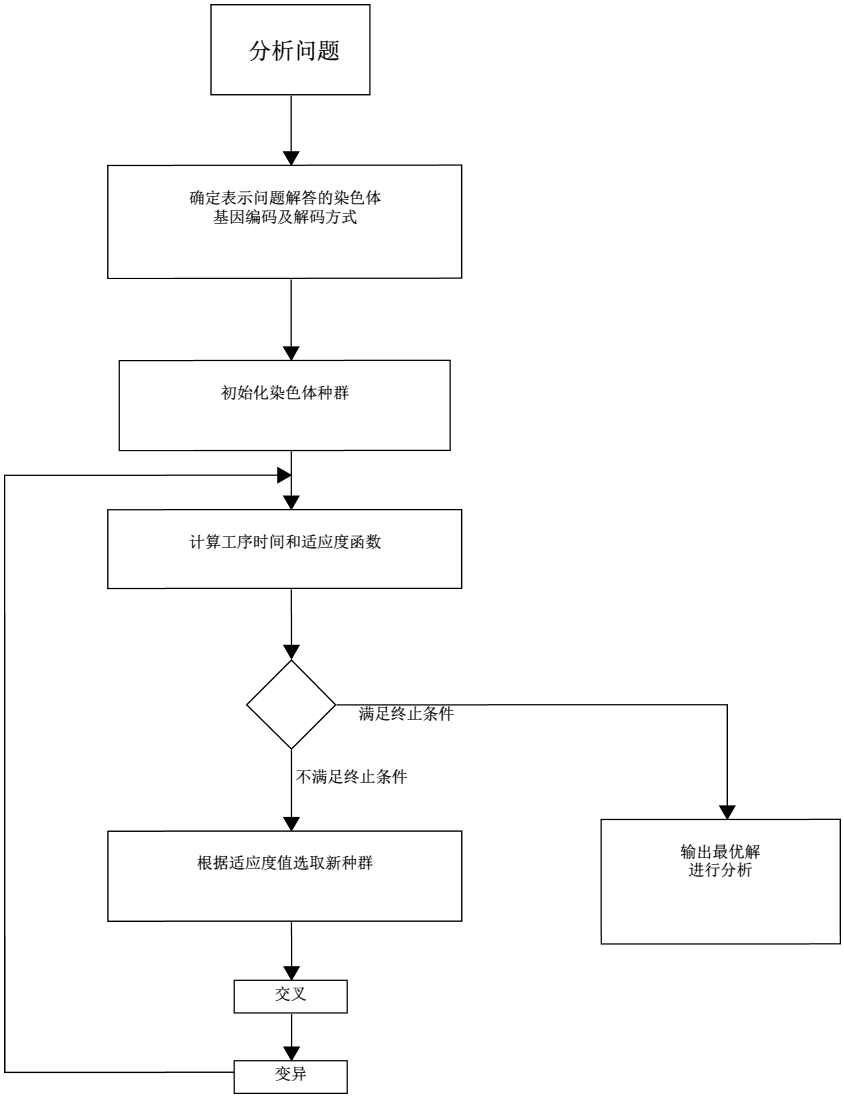


图 5: 程序执行流程图

表 3: 种群适应性与迭代次数关系表

| 平均适应度值 模拟次数 / 迭代次数 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 5 | 117.8 | 118.6 | 116.4 | 105.53 | 115.33 | 110.86 | 107.66 | 116.6 |
| 10 | 105.4 | 105.8 | 101.13 | 106.33 | 101.4 | 113.93 | 98.2 | 110.6 |
| 15 | 101.66 | 101 | 98.46 | 100.46 | 102 | 102.6 | 98.2 | 98.73 |
| 20 | 99 | 97 | 97 | 98.86 | 98.2 | 102.46 | 97 | 97 |
| 25 | 99 | 97 | 97 | 98.86 | 99.86 | 98.46 | 101 | 97 |
| 30 | 99 | 97 | 97 | 98.86 | 97 | 97 | 97 | 97 |

程序的主要参数如下:

1. 初始种群的规模为 $N = 15$
2. 选择操作中精英保留策略适应度平均值参数 $\alpha = 1.3$
3. 自适应交叉概率的初始值为 $p_{c0} = 0.9$，交叉概率成长值 $p_{cst} = 0.1$
4. 自适应变异概率的初始值为 $p_{m0} = 0.02$，变异概率成长值为 $p_{mst} = 0.01$

我们通过如下几个方面来评估算法的运行效果。

1. 在种群大小 $N$ 一定的情况下，每一代种群的适应度函数平均值与迭代次数的关系。

   根据表 3做出折线图 6。

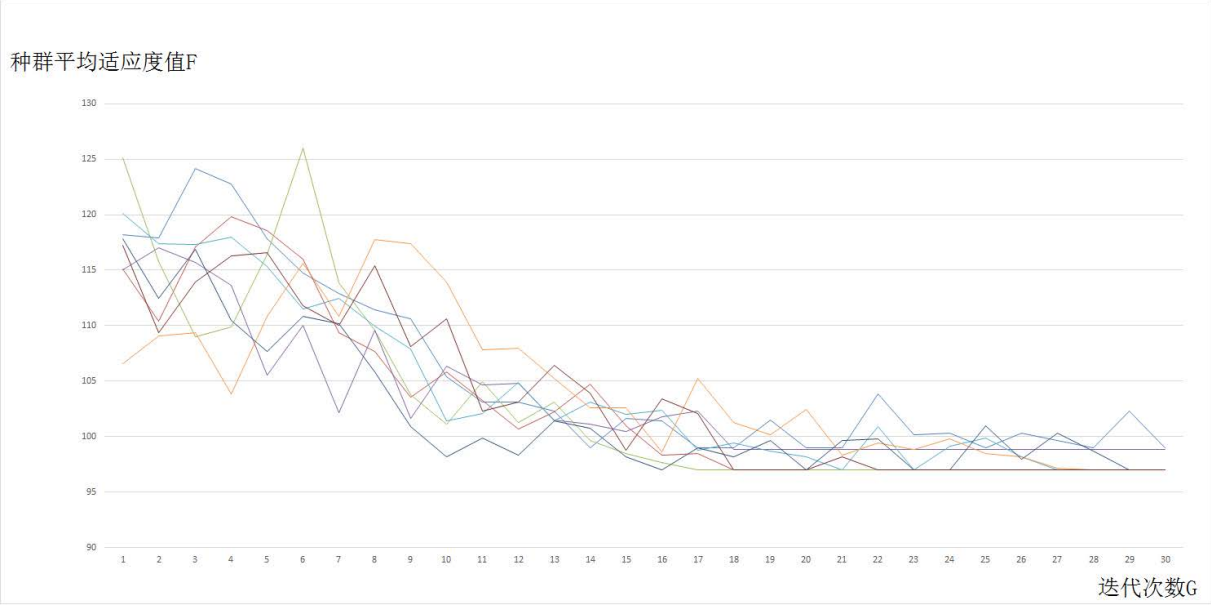


图 6: 种群适应性与迭代次数关系图

在图 6中的 8 次模拟中，有 6 次模拟都在进行了 20 次迭代左右便趋向于稳定，对问题的目标得到最优解 97。因此算法的稳定性还是比较高的，而且在迭代 30 次时已经达到最优解，说明算法的效率也比较高。

表 4: 种群适应性与初始种群规模关系表

| 种群平均适应度值 模拟次数 / 种群初始规模 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 15 | 98.4667 | 103.4667 | 98.7333 | 101.2 | 97 |
| 20 | 98.4 | 99 | 98.9 | 99 | 97 |
| 25 | 97 | 99 | 98.76 | 98.76 | 97 |
| 30 | 97 | 99 | 98.4667 | 98.1333 | 97 |
| 35 | 97 | 99 | 97.5143 | 97.6857 | 97 |
| 40 | 97 | 99 | 97 | 97.6 | 97 |

2. 在迭代次数一定的情况下，种群的适应度函数值与初始种群的规模的关系。同样的，根据表 4我们做出折线图 7。

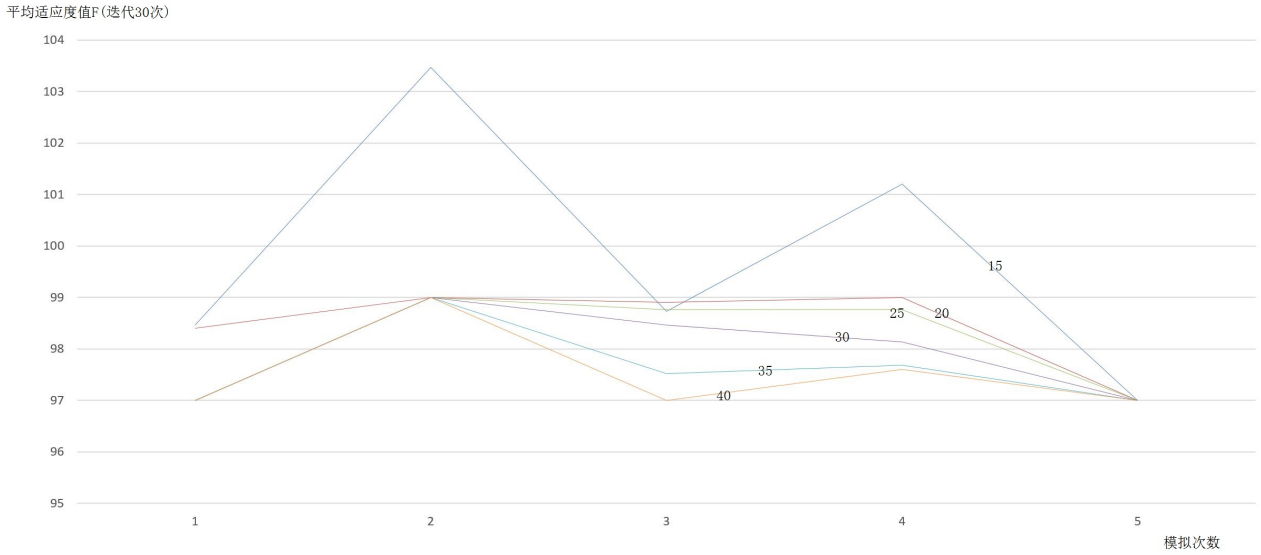


图 7: 种群适应性与初始种群规模关系图

从表 3和图 7中很容易看出，随着初始种群数目的增大，遗传算法迭代 30 次后平均的适应度值更加靠近最优解 97，适当地选择种群规模可以大大提高算法的准确性。

## 6 总结评价

本文通过实现遗传算法的各个操作过程，解决了问题中最小化最大完工时间的问题，而且在迭代 30 次时就可以得到较为稳定的最优解，因此算法的稳定性和效率还都不错，并且可以解决超过问题中工件个数的工件加工问题。

但是通过实例求解也可以看出，在迭代的后期，特别是初始种群规模比较小的时候，还是有一定概率陷入局部最优解 99 中，从而得不到最终的正确结果，而且该算法不能处理加工某道工序的可用机器超过一台的情况。

## 参考文献

[1] 王旭东. 一种基于改进遗传算法的柔性流水车间调度问题研究 [D]. 广东工业大学.2014.

[2] 吴波. 基于遗传算法的作业车间调度问题研究 [D]. 大连交通大学,2005.

# A 具体程序实现

## A.1 数组定义

```
#define MAXN 200
int a[MAXN];//用于记录每个工件的工序数
int pos_1[MAXN];//用于交叉操作临时保存子染色体的序列
int pos_2[MAXN];
int tmp_1[MAXN];//用于变异操作临时保存子染色体的序列
int tmp_2[MAXN];
int s[MAXN][MAXN];//用于保存种群的染色体
int tmp[MAXN][MAXN];//在选择操作时暂时存储染色体
struct det{
    int m;
    int t;
};//记录工序所用的机器和时间
struct det p[MAXN][MAXN];//用于记录每道工序的具体信息
int pt[MAXN][MAXN];//用于记录每道工序完成时该工件的总耗时
int ma[MAXN];//用于记录完成某道工序时该机器的耗时
int cnt[MAXN];//在计算完工时间时记录某工件的工序是否全部完成
int total[MAXN];//用于记录染色体的完工时间
double suit[MAXN];//用于记录染色体的适应度函数
double prob[MAXN];//用于记录根据适应度值各染色体被选中的概率
bool chos[MAXN];//用于在选择操作中记录某染色体是否被选中
```

## A.2 *produceChrom*() 函数定义

n 为工件个数，len 为染色体的长度，m 为种群中染色体的个数

```
void produceChrom(int a[],int n,int len,int m) {
    memset(s,0,sizeof(s));
    for(int i=1;i<=m;i++) {
        srand((int)time(0)+rand());
        for(int j=1;j<=n;j++) {
            int tmp=0;
            while(tmp!=a[j]) {
                int index=rand()%len+1;
                if(!s[i][index]) {s[i][index]=j;tmp++;}
            }
        }
    }
}
```

### A.3 *cal_time*() 和 *cal_suit*() 函数定义

n 为工件个数，len 为染色体长度，*pos* 为选中的染色体在在种群中的位置

```
void cal_time(struct det p[][MAXN],int a[],int n,int s[][MAXN],int len,
    int pos,int total[]) {
        memset(ma,0,sizeof(ma));
        memset(pt,0,sizeof(pt));
        for(int i=1;i<=n;i++) cnt[i]=1;
        for(int i=1;i<=len;i++) {
            int part=s[pos][i];
            int seq=cnt[part];
            int mac=p[part][seq].m;
            int time=p[part][seq].t;
            pt[part][seq]=max(ma[mac],pt[part][seq-1])+time;
            ma[mac]=pt[part][seq];
            if(seq==a[part]) if(total[pos]<pt[part][seq]) total[pos]=pt[
                part][seq];
            cnt[part]++;
        }
}
void cal_suit(int n,int s[][MAXN],int len,int pos,int total[]) {
    cal_time(p,a,n,s,len,pos,total);
    suit[pos]=1.0/total[pos];
}
```

### A.4 *selectChrom*() 函数定义

n 为工件个数，len 为染色体长度，tot 为当前种群规模，m 为要选择的染色体个数

```
void selectChrom(int n,int s[][MAXN],int len,int tot,int m,int total[]) {
    double sum=0;
    for(int i=1;i<=tot;i++) {
        sum+=suit[i];
    }
    memset(chos,0,sizeof(chos));
    prob[0]=0;
    for(int i=1;i<=tot;i++) prob[i]=suit[i]/sum;
    for(int i=1;i<=tot;i++) {prob[i]+=prob[i-1];}
    srand((int)time(0)+rand());
    int cnt=0;
    double average=sum*1.3/tot;
```

```
    for(int i=1;i<=tot&&cnt<m;i++) {
            if(suit[i]>=average) {
                chos[i]=true;
                cnt++;
            }
        }
    while(cnt<m) {
        double ran=rand()*1.0/RAND_MAX;
        for(int i=1;i<=tot;i++) {
            if(ran>=prob[i-1]&&ran<prob[i]) {
                if(chos[i]==0) {
                    chos[i]=true;
                    cnt++;
                    break;
                }
            }
        }
    }
    cnt=0;
    memset(tmp,0,sizeof(tmp));
        for(int i=1;i<=tot&&cnt<m;i++) {
            if(chos[i]) {
                cnt++;
                for(int j=1;j<=len;j++) {tmp[cnt][j]=s[i][j];}
            }
        }
    memset(s,0,sizeof(s));
    for(int i=1;i<=m;i++) {
        for(int j=1;j<=len;j++) {
            s[i][j]=tmp[i][j];
        }
    }
    for(int i=1;i<=m;i++) total[i]=0;
    for(int i=1;i<=m;i++) {
        cal_suit(n,s,len,i,total);
    }
}
```

### A.5 *crossOverChrom*() 函数定义

len 为染色体的长度，pos1 和 pos2 为参与交换的两条父染色体在种群中的位置，tot 为当前种群规模

```
void crossover(int s[][MAXN],int len,int pos1,int pos2,int &tot) {
    srand((int)time(0)+rand());
    for(int i=1;i<=len;i++) {
        tmp_1[i]=s[pos1][i]; tmp_2[i]=s[pos2][i];
    }
    int ran=rand()%len;
    for(int i=1;i<=ran;i++) {
        pos_1[i]=tmp_1[i]; pos_2[i]=tmp_2[i];
    }
    int cnt=1;
    while(cnt<=ran) {
        for(int i=1;i<=len&&cnt<=ran;i++) {
            if(pos_2[cnt]==tmp_1[i]) {
                tmp_1[i]=0; cnt++;
            }
        }
    }
    cnt=1;
    while(cnt<=ran) {
        for(int i=1;i<=len&&cnt<=ran;i++) {
            if(pos_1[cnt]==tmp_2[i]) {
                tmp_2[i]=0; cnt++;
            }
        }
    }
    cnt=ran+1;
    for(int i=1;i<=len;i++) {
        if(tmp_1[i]) {
            pos_2[cnt]=tmp_1[i]; cnt++;
        }
    }
    cnt=ran+1;
    for(int i=1;i<=len;i++) {
        if(tmp_2[i]) {
            pos_1[cnt]=tmp_2[i]; cnt++;
        }
```

```
    }
    for(int i=1;i<=len;i++) {
        s[tot+1][i]=pos_2[i];
        s[tot+2][i]=pos_1[i];
    }
    tot+=2;
}
```

### A.6 *mutateChrom*() 函数定义

len 为染色体的长度，pos 为发生变异的父染色体在种群中的位置，tot 为当前种群规模

```
void mutate(int s[][MAXN],int len,int pos,int &tot) {
    srand((int)time(0)+rand());
    for(int i=1;i<=len;i++) tmp_1[i]=s[pos][i];
    int ran1=rand()%len+1;
    int ran2=rand()%len+1;
    while(ran1==ran2) ran2=rand()%len+1;
    if(ran1>ran2) {int t=ran1;ran1=ran2;ran2=t;}
    memcpy(tmp_2,tmp_1,sizeof(tmp_1));
    for(int i=ran1;i<=ran2;i++) {
        tmp_2[i]=tmp_1[ran1+ran2-i];
    }
    memcpy(s[tot+1],tmp_2,sizeof(tmp_2));
    tot++;
}
```

### A.7 *manipulateChrom*() 函数定义

n 为工件个数，len 为染色体长度，num 为种群规模，$p_{c0}, p_{cst}, p_{m0}, p_{mst}$ 为交叉和变异相关概率值，$avg_{suit}$ 为初代种群的适应度平均值

```
void manipulate(int n,int s[][MAXN],int len,int num,int total[],double
    suit[],double pc0,double pcst,double pm0,double pmst,double avg_suit)
    {
    srand((int)time(0)+rand());
    double suit_avg=0;
    for(int i=1;i<=num;i++) suit_avg+=suit[i];
    suit_avg/=num;
    double pc=pc0-(suit_avg-avg_suit)*pcst/avg_suit;
    int pair=(int)(num*pc);
    printf("%.4f\n",suit_avg);
```

```
    int tot=num;
    for(int i=1;i<=pair;i++) {
        int ran1=rand()%num+1;
        int ran2=rand()%num+1;
        while(ran1==ran2) ran2=rand()%num+1;
        crossOverChrom(s,len,ran1,ran2,tot);
    }
    for(int i=num+1;i<=tot;i++) {
        cal_suit(n,s,len,i,total);
    }
    memset(prob,0,sizeof(prob));
    for(int i=1;i<=tot;i++) {
        if(suit[i]<=suit_avg) prob[i]=pm0;
        else prob[i]=pm0-(suit[i]-suit_avg)*pmst/suit_avg;
    }
    for(int i=1;i<=tot;i++) prob[i]+=prob[i-1];
    int tmp_tot=tot;
    for(int i=1;i<=num;i++) {
        double ran=rand()*1.0/RAND_MAX;
        for(int j=1;j<=tmp_tot;j++) {
            if(ran>=prob[j-1]&&ran<=prob[j]) {
                mutateChrom(s,len,j,tot);
                break;
            }
        }
    }
    for(int i=tmp_tot+1;i<=tot;i++) {
        cal_suit(n,s,len,i,total);
    }
    selectChrom(n,s,len,tot,num,total);
}
```