遗传算法简要报告

董若扬　学号：1120202944

1　简介

吸取模拟退火算法设计初期考虑不周顶层设计欠缺的教训，设计遗传算法之初就采用面向对象的设计思想和规范进行设计。大致设计如下：

a) **Main 类：**

i. 字段：

1. 工件数 *Row_Workpiece*，机器数 *Column_Machine*
2. 演化代数 *GENERATION*
3. 保留概率，交换概率，变异概率，自交概率

ii. 方法：

1. 进行数据的读取
2. 创建数据对象
3. 数据操作
4. 数据输出

b) **Chromosome 染色体类：**

i. 字段：

1. 基因长度（解空间大小）*geneLength*　基因深度（可能含有多位信息）*geneDepth*
2. 原始适应值 rawFitness 标定适应值 calibratedFitness

ii. 方法：

1. 计算原始适应值 `void calculateRawFitness()`
2. 循环交叉互换 `static void cycleCrossover(Chromosome p1, Chromosome p2)`
3. 随机片段交叉互换 `static void orderCrossover(Chromosome p1, Chromosome p2)`
4. 微生物仿真交叉互换 `static void microbialCrossover(Chromosome p1, Chromosome p2)`
5. 染色体异位变异 `void ectopicMutate()`

6. 染色体倒位变异 `void reversedMutate()`
7. 染色体剪切拼接变异 `void shearMutate()`
8. 染色体重置变异 `void resetMutate()`
9. 打印染色体信息 `static void show(List<Chromosome> chromosomeList)`

c) Population 种群类

i. 字段：

1. 种群数量 *NUMBER_OF_SPECIES*
2. 波动值 *epsilon*
3. 波动值衰减系数 *ITERATION*
4. 种群最大原始适应值 *maxFitnessValue*
5. 标定适应值之和 *calibratedFitnessSum*
6. 适应值转盘 *turntable*
7. 种群容器 *chromosomes* 种群容器副本 *chromosomesSaved*
8. 父本 *parentIndex1* 母本 *parentIndex2*

ii. 方法：

1. 搜索并保存最大初始适应值 `static void findMaxFitnessValue(List<Chromosome> chromosomeList)`
2. 标定原始适应值 `static void calibrateFitness(List<Chromosome> chromosomeList)`
3. 计算标定适应值之和 `static void calculateCalibratedFitnessValueSum(List<Chromosome> chromosomeList)`
4. 根据标定适应值生成转盘 `static void generateTurntable(List<Chromosome> chromosomeList)`
5. 旋轮法选择双亲 `static void selectParents()`

2 实现

2.1 编码方式

采用顺序编码，即 0，1，2，……，M-1（M 为机器数量），初始化时令数组每个单元的值为其索引值，再使用随机数算法打乱。

```
public void initGeneData() {
    for (int i = 0; i < geneLength; i++) {
        geneData[i] = i;
    }
    for (int i = geneLength; i > 0; i--) {
        swap(random.nextInt(i), i - 1);
    }
}
```

2.2 适值函数

因为该题要求的是最小值，以最小消耗时间作为问题的适值函数无法使后代的选择概率同适应值的好坏成正相关关系，所以需要标定。

标定前先计算问题本身需要的解，即任意顺序下最小加工时间。

```
public void calculateRawFitness() {
    int[][] dp     = new int[geneLength][geneDepth];
    int[][] values = Main.timeTab;

    dp[0][0] = values[geneData[0]][0];

    for (int i = 1; i < geneLength; i++) {
        dp[i][0] = dp[i - 1][0] + values[geneData[i]][0];
    }
    for (int j = 1; j < geneDepth; j++) {
        dp[0][j] = dp[0][j - 1] + values[geneData[0]][j];
    }
    for (int i = 1; i < geneLength; i++) {
        for (int j = 1; j < geneDepth; j++) {
            dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]) +
values[geneData[i]][j];
        }
    }
    rawFitness = dp[geneLength - 1][geneDepth - 1];
}
```

计算完种群中每个染色体的适应值后，保存最大适应值 `public static int maxFitnessValue;`

对所有原始适应值进行标定，具体过程为种群最大适应值-个体适应值+常数补偿

```
public static void calibrateFitness(List<Chromosome> chromosomeList)
{
    for (Chromosome ch : chromosomeList) {
        ch.setCalibratedFitness(maxFitnessValue - ch.getRawFitness() +
(int) epsilon);
    }
}
```

这样，个体适应值与解的优劣关系成正相关关系，且适应值之间差距小，不容易拉开悬殊的差距（例如使用反比例函数进行标定，标定值随着原始值的增加快速减小），并且添加了常数项补偿，使得最坏的那些染色体上的信息仍然有几率得以保留，常数值会随着遗传的进行，逐渐减小，以增加自然选择力，即后期最坏的染色体得到补偿的机会会逐渐减小。

2.3 旋轮法选择双亲

生成转盘，完成了以下行为：*1.寻找最大初始适应值;2.标定适应值;3.生成转盘;4.计算标定适应值之和.*

```
public static void generateTurntable(List<Chromosome> chromosomeList)
{
    findMaxFitnessValue(chromosomeList);
    calibrateFitness(chromosomeList);
    int value = 0;
    for (int i = 0; i < chromosomeList.size(); i++) {
        value += chromosomeList.get(i).getCalibratedFitness();
        turntable[i] = value;
    }
    calibratedFitnessSum = value;
}
```

根据这一代的转盘，根据旋轮法正比例选择双亲。在 0 到适应值之和内生成一个随机数，看此随机数落在转盘的几号索引的区间内，染色体容器处相同索引处的染色体即为选中的亲本，每次交叉互换选择两次。

```
public static void selectParents() {
    int rand1 = random.nextInt(calibratedFitnessSum);
    int rand2 = random.nextInt(calibratedFitnessSum);
    parentIndex1 = 0;
    parentIndex2 = 0;
```

```
    /* 查找双亲 */
    boolean parent1Found = false;
    boolean parent2Found = false;
    while (!(parent1Found && parent2Found)) {
        if (!parent1Found && turntable[parentIndex1] < rand1) {
            parentIndex1++;
        } else {
            parent1Found = true;
        }
        if (!parent2Found && turntable[parentIndex2] < rand2) {
            parentIndex2++;
        } else {
            parent2Found = true;
        }
    }

}
```

2.4 遗传算子（交叉互换）

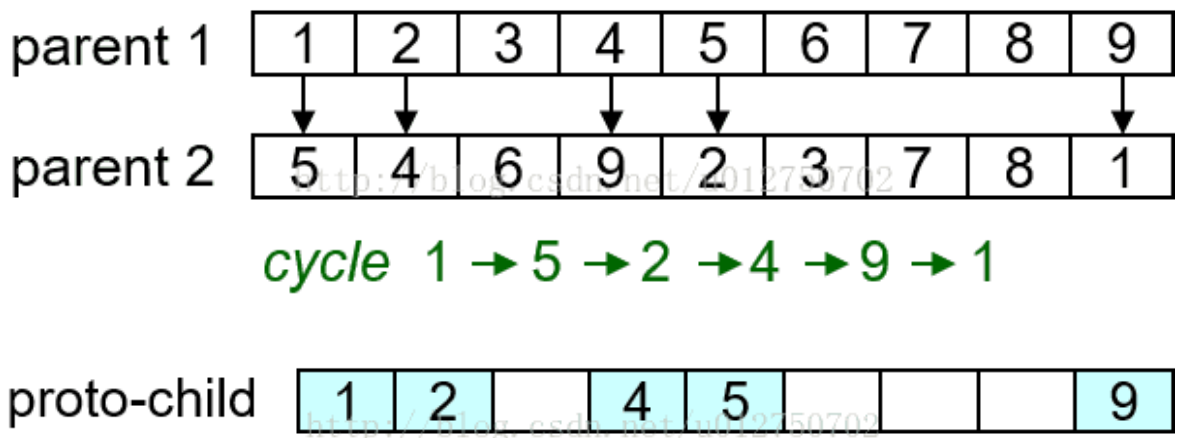
2.4.1 循环交叉法：

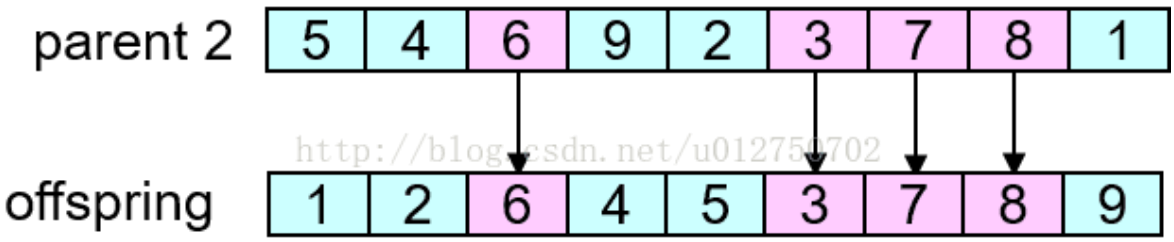
（1） 选 P1 的第一个元素作为 C1 的第一位，选 P2 的第一个元素作为 C2 的第一位。

（2） 到 P1 中找 P2 的第一个元素赋给 C1 的相应位置……重复此过程，直到 P2 上得到 P1 的第一个元素为止，称为一个循环。

（3） 对最前的基因按 P1，P2 基因轮替原则重复以上过程。

（4） 重复以上过程，直到所有位都完成。






```
public static void cycleCrossover(Chromosome parent1, Chromosome
parent2) {
    /* KEY: value, VALUE: index. */
    HashMap<Integer, Integer> map1 = new HashMap<>(geneLength);
    for (int i = 0; i < geneLength; i++) {
        map1.put(parent1.geneData[i], i);
    }
    Chromosome child1   = new Chromosome();
    Chromosome child2   = new Chromosome();
    boolean[]  visited  = new boolean[geneLength];
    boolean    onesTurn = true;
    /* 父代染色体交叉交换 */
    for (int i = 0; i < geneLength; i++) {
        if (visited[i]) {
            continue;
        }
        int j = i;
        do {
            visited[j] = true;
            if (onesTurn) {
                child1.geneData[j] = parent1.geneData[j];
                child2.geneData[j] = parent2.geneData[j];
            } else {
                child1.geneData[j] = parent2.geneData[j];
                child2.geneData[j] = parent1.geneData[j];
            }
            j = map1.get(parent2.geneData[j]);
        } while (j != i);
        onesTurn = !onesTurn;
```

```
    }

    child1.calculateRawFitness();

    child2.calculateRawFitness();

    Population.chromosomes.add(child1);
    Population.chromosomes.add(child2);

}
```

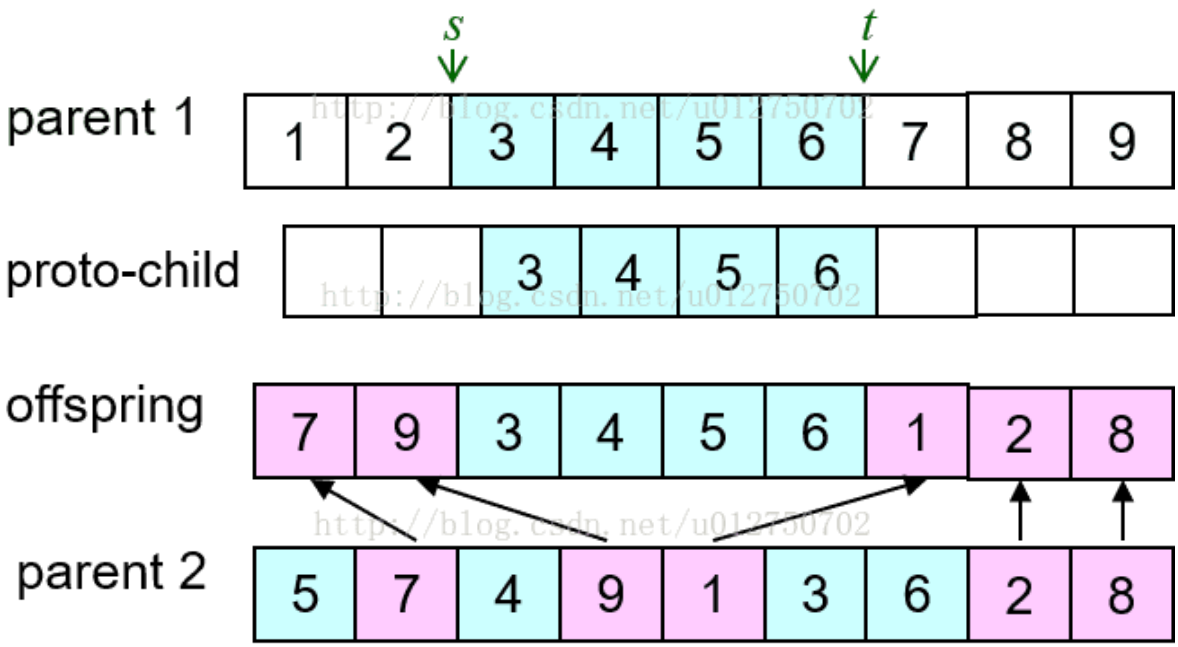
2.4.2 顺序交叉法

（1） 随机选切点 X,Y；

（2） 交换中间部分；

（3） 从切点 Y 后第一个基因起列出原序，去掉已有基因；

（4） Y 后第一个位置起填入。




```
public static void orderCrossover(Chromosome parent1, Chromosome
parent2) {
/*
 染色体交换片段
```

```
 Order Crossover (OX)
 若不控制随机数的值,有 p= 2/(geneLength)^2 的概率不发生交换, 即完全保留亲本.
*/
    int     rand1, rand2;
    boolean isNotValidExchange;
    do {
        rand1 = random.nextInt(geneLength);
        rand2 = random.nextInt(geneLength - rand1) + rand1;
        isNotValidExchange = (rand1 == 0 && rand2 == geneLength) ||
(rand1 == rand2 && parent1.geneData[rand1] ==
parent2.geneData[rand2]);
    } while (isNotValidExchange);

    Chromosome child1 = new Chromosome();
    Chromosome child2 = new Chromosome();

    boolean[] visited1 = new boolean[geneLength];
    boolean[] visited2 = new boolean[geneLength];

    for (int i = rand1; i <= rand2; i++) {
        child1.geneData[i] = parent1.geneData[i];
        visited1[parent1.geneData[i]] = true;
        child2.geneData[i] = parent2.geneData[i];
        visited2[parent2.geneData[i]] = true;
    }

    int target1 = 0, target2 = 0;

    for (int i = 0; i < geneLength; i++) {
        while (target1 >= rand1 && target1 <= rand2) {
            target1++;
        }
        while (target2 >= rand1 && target2 <= rand2) {
            target2++;
        }
```

```
        if (!visited1[parent2.geneData[i]]) {
            child1.geneData[target1] = parent2.geneData[i];
            target1++;
        }
        if (!visited2[parent1.geneData[i]]) {
            child2.geneData[target2] = parent1.geneData[i];
            target2++;
        }
    }
    /* 考虑变异 */

    child1.calculateRawFitness();
    child2.calculateRawFitness();

    Population.chromosomes.add(child1);
    Population.chromosomes.add(child2);

}
```

2.4.3 微生物遗传算法

取出两个染色体，比较适应值大小，适应值大的保留，适应值小的采用任一种交叉算法生成一个子代，适应值大的亲本和子代一同作为下一代。此处采用循环交叉法。

```
public static void microbialCrossover(Chromosome parent1, Chromosome parent2) {/* p1 适应值高, p2 适应值低 */
        HashMap<Integer, Integer> map1 = new HashMap<>(geneLength);
        for (int i = 0; i < geneLength; i++) {
            map1.put(parent1.geneData[i], i);
        }
        boolean[] visited = new boolean[geneLength];

        int index = 0;
        do {
            visited[parent1.geneData[index]] = true;
            parent2.geneData[index] = parent1.geneData[index];

            index = map1.get(parent2.geneData[index]);
```

```
        } while (index != 0);

        int target=0;
        for (int i = 0; i < geneLength; i++) {
            if (!visited[parent2.geneData[i]]) {
                parent2.geneData[target] = parent2.geneData[i];
                target++;
            }
        }

        /* 考虑变异 */

        parent1.calculateRawFitness();
        parent2.calculateRawFitness();
        Population.chromosomes.add(parent1);
        Population.chromosomes.add(parent2);

    }
}
```

2.5 遗传算子（变异）

2.5.1 染色体随机交换片段

模拟基因变异中的交换

```
public void ectopicMutate() {
    final double ECTOPIC_POSSIBILITY = 0.5;
    do {
        int random1 = rand.nextInt(geneLength);
        int random2 = rand.nextInt(geneLength);
        swap(random1, random2);
    } while (rand.nextDouble() < ECTOPIC_POSSIBILITY);
}
```

2.5.2 染色体倒转

模拟染色体变异中的倒位变异

```
public void reversedMutate() {
    for (int i = 0; i < geneLength / 2; i++) {
        swap(i, geneLength - 1 - i);
```

```
    }
}
```

2.5.3 剪切染色体后重新拼接

模拟染色体变异中的异位变异

```
public void shearMutate() {
    int cutPoint;
    do {
        cutPoint = rand.nextInt(geneLength);
    } while (cutPoint == 0);
    int[] geneDataSaved = geneData;
    geneData = new int[geneLength];
    int index = 0;
    for (int i = cutPoint; i < geneLength; i++) {
        geneData[index++] = geneDataSaved[i];
    }
    for (int i = 0; i < cutPoint; i++) {
        geneData[index++] = geneDataSaved[i];
    }

}
```

2.5.4 随机重置染色体

模拟染色体受外在物理如辐射，化学如诱导试剂，生物如病毒等因素影响产生的突变

```
public void resetMutate() {
    for (int i = geneLength; i > 0; i--) {
        swap(random.nextInt(i), i - 1);
    }
}
```

3 未完成部分

i. 时间匆忙，完成了基本功能和较多的遗传算法算子，参数还没调节，目前优化效果不理想，不仅比模拟退火算法消耗更多的时间空间，而且解的质量不如模拟退火算法。

ii. 未重构成多线程形式，执行效率有待提高。

iii. 设计了很多交叉互换，变异的算子，仍有大量算子还未实现，包括自交等特殊情况。

iv. 生物学上知识遗忘了，为了创新地移植某些概念进来，还得学习。

v. 之所以写遗传算法，目的并不是为了和模拟退火算法做比较，而是上课的时候发现这玩意有点像在计算机上研究自然，研究社会，特别是不同的筛选机制，不同的适值函数，不同的选择压力，不同的遗传算子都给了模拟社会运转极大的自由性，我的构想是设计几套不同的模式，模拟比如“个人英雄主

义”，“社会达尔文主义”，“纳粹主义”等社会意识形态的表现。

vi. 还没有找到“主义”的恰当译名

以上部分，将在这个暑假实现。