

遗传算法实验报告

一、实验需求

1. 采用遗传算法 `Genertic Algorithm` 求解TSP问题，要求采取与模拟退火求解相同的问题规模，设计算法步骤，实现可视化求解问题过程。
2. 设计较好的交叉操作，并且引入多种局部搜索操作(可以通过替换遗传算法的变异操作实现)，获得较优解，与之前的模拟退火算法求解进行比较，得出比较结果。
3. 得出设计高效遗传算法的一些经验，并且比较单点搜索与多点搜索的优缺点。

二、实验环境

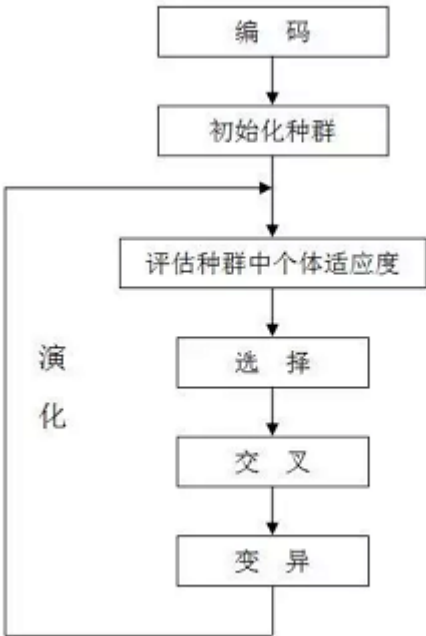
- 操作系统: `Windows10`
- 实现可视化编程: 前端 `Bootstrap + JS` , 服务端 `NodeJS`
- 后端编程语言: `C++`

三、实验过程

(1) 遗传算法求解TSP问题的后端实现

编程思想: 面向对象编程(OOP)

基本思想: 遗传算法



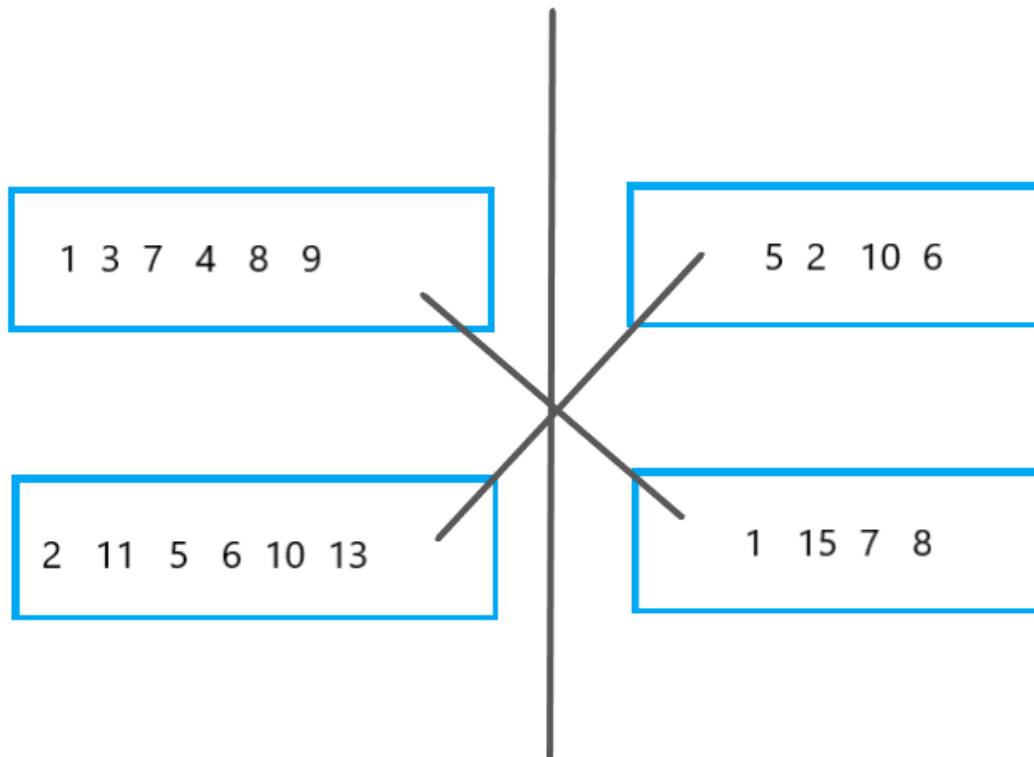
基本架构:

1. 定义求解TSP问题的基本遗传算法类定义 `GA`，该类内部包含各项种群操作函数：交叉方法 `cross()`，变异方法 `varia`，优良性状保留方法 `holdGoodPath()`，轮盘赌选择操作个体方法 `choose()`。将这些内部方法定义为内联函数 `inline`，适当加快内部函数调用速度。对于外部引用，只暴露对应的 `getShortestPath()` 函数，直接返回遗传算法根据对应步数迭代，求解完成之后的结果路径。

```
class GA{
public:
    GA();
    Path getShortestPath();
    void clearPath(){
        for(int i=0; i<GROUP_SIZE; i++){
            group[i] = Path();
        }
    }

private:
    // All classified population group
    vector<Path> group;
    Path bestPath;
    // Private inline methods
    inline void choose(vector<Path>& oth);
    inline void cross(vector<Path>& oth);
    inline void varia(vector<Path>& oth);
    inline void holdGoodPath(vector<Path>& old, vector<Path>& gr);
    inline void writePathFile(Path& in);
};
```

2. `GA` 求解问题依赖于对个体之间的基本操作获取具有更好性状的后代，此处的个体在TSP问题中对应于路径 `Path`，故此处定义相关路径结构体是十分必要的。
 - 路径结构体 `Path` 内部保存当前路长度 `length`，以及路径上的城市编号队列 `path`。构造函数使用标准库 `STL` 内的 `random_shuffle` 方法，随机生成初始路径节点队列。
 - 结构体内部仍定义了计算当前节点长度函数 `calculateLength()`，实际上即为根据规模输入文件，求每两点之间距离然后求和即可。此处的个体交叉操作采用了单点交叉操作 `singlePointCross()`。
 1. 单点交叉为随机选取序列上的某一点，以该点前后，分别对两个个体的路径进行交叉交换操作。
 2. 同时为了保证交叉操作的合法性，建立 `Flag` 数组保留两个个体的路径编号交叉状态(True or False)，使得结果路径上不出现重复非法节点。



- 变异操作 `randomVariation()` 结合了模拟退火法的邻域交换方法，此处根据获得的随机数，进行不同变异方法的选取：

1. 基本的单点交换
2. 选取路径段范围的节点队列反转
3. 选取路径段范围的循环左移

```
struct Path{
    // Path's basic infos
    double length;
    int path[CITY_NUMBER];
    // Path's behavior methods
    Path(){
        length = 0;
        // Initial random generation
        generate(path, path + CITY_NUMBER, linearGen(0));
        random_shuffle(path, path + CITY_NUMBER); // Randomly shuffle
        // Calculate init length
        calculateLength();
    }

    void initDistance(){
        vector<Node> cityNodes;

        ifstream in("src/cities.txt");
        for(int i=0; i<CITY_NUMBER; i++){
            cityNodes.push_back(Node());
            in >> cityNodes[i].num >> cityNodes[i].x >> cityNodes[i].y;
        }

        for(int i=0; i<CITY_NUMBER; i++){
```

```

        distanceTable[i][i] = (double)INT_MAX;
        for(int j=i+1; j<CITY_NUMBER; j++){
            double dist = sqrt((cityNodes[i].x - cityNodes[j].x) * (cityNodes[i].x -
cityNodes[j].x) +
                (cityNodes[i].y - cityNodes[j].y) * (cityNodes[i].y -
cityNodes[j].y));
            distanceTable[i][j] = distanceTable[j][i] = dist;
        }
    }
}

// Calculating path's length
void calculateLength(){
    length = 0;
    // Iteration on path
    if(distanceTable[0][0] == 0.0){
        initDistance();
    }
    for(int i=1; i<CITY_NUMBER; i++){
        length += distanceTable[path[i-1] - 1][path[i]-1];
    }
    // Don't forget to add first and last city's distance
    length += distanceTable[path[CITY_NUMBER-1] - 1][path[0] - 1];
}

// Single Point CrossOver behavior
void singlePointCross(Path& oth){
    int randomMark = rand() % (CITY_NUMBER - 2) + 1;
    // Swapping with another population
    for(int i=randomMark; i<CITY_NUMBER; i++){
        int temp = path[i];
        path[i] = oth.path[i];
        oth.path[i] = temp;
    }
    // Establish two population's corss state array
    int index = 0, oth_index = 0;
    bool crossFlag[CITY_NUMBER + 1] = { false };
    bool oth_crossFlag[CITY_NUMBER + 1] = { false };
    // Start two path's single point cross
    while(index < CITY_NUMBER && oth_index < CITY_NUMBER){
        if(crossFlag[path[index]] && oth_crossFlag[oth.path[oth_index]]){
            int temp = path[index];
            path[index] = oth.path[oth_index];
            oth.path[oth_index] = temp;
            ++index;
            ++oth_index;
        }
        // Judge whether reach depth
        if(index >= CITY_NUMBER || oth_index >= CITY_NUMBER)
            break;
        if(!crossFlag[path[index]]){
            crossFlag[path[index]] = true;

            ++index;

```

```

    }
    if(!oth_crossFlag[oth.path[oth_index]]){
        oth_crossFlag[oth.path[oth_index]] = true;
        ++oth_index;
    }
}
calculateLength();
oth.calculateLength();
}

// Variation behavior
void randomVariation(){
    int a = rand() % CITY_NUMBER, b = rand() % CITY_NUMBER;
    if(a > b){
        swap(a, b);
    }else if(a == b){
        return;
    }
    // Switch random variation type
    int target = rand() % 3;
    switch(target){
        case 0:
            swap(path[a], path[b]);
            break;
        case 1:
            reverse(path + a, path + b);
            break;
        default:
            if(b < CITY_NUMBER - 1){
                rotate(path + a, path + b, path + b + 1);
            }
    }
    calculateLength();
}

};

```

3. 通过阅读相关文献，结合老师所给资料，将相关的变异、交叉等概率进行定义，同时约束种群大小，以及整体迭代次数。

```

#define VARIA_PRO 0.5
#define CROSS_PRO 0.8
#define CITY_NUMBER 130
#define GROUP_SIZE 500
#define ITER_TIME 8000

```

重点函数具体实现：

1. 根据当前定义迭代总轮数，进行基于循环的种群内部操作，产生当前种群的后代。每一轮迭代需要依次进行个体选取，以及相应的交叉、变异操作，进行完相关方法之后保留父代中优良个体，维持到子代种群。

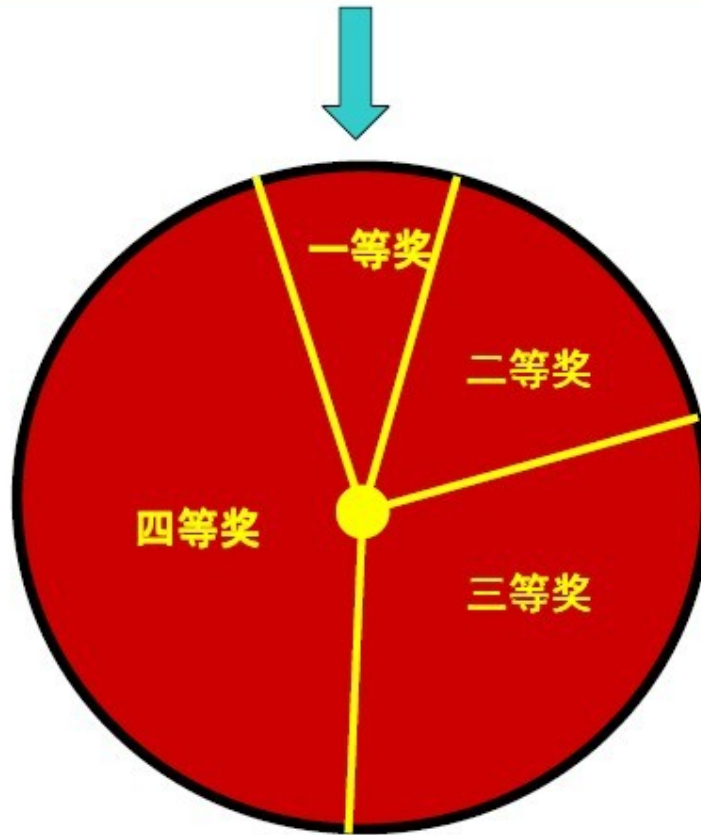
```
Path GA::getShortestPath(){
    srand((unsigned)time(NULL));

    Path bestResult;

    for(int i=0; i<ITER_TIME; i++){
        vector<Path> prevGroup = group;
        // Populations choose, cross, variation
        choose(group);
        cross(group);
        for(int j=0; j<5; j++){
            varia(group);
            holdGoodPath(prevGroup, group);
        }
        for(int j=0; j<GROUP_SIZE; j++){
            if(group[j].length < bestResult.length){
                bestResult = group[j];
            }
        }
    }
    writePathFile(bestResult);
    bestResult.calculateLength();
    return bestResult;
}
```

2. 采取轮盘赌方法进行个体选择，依据每个个体的适应度(实际上就是个体路径内部的长度倒数)，适应度越低的个体越难被选做遗传算法的操作个体，反之适应度越高，即性状越优良，选为产生后代的父代个体的可能性越大。

轮盘赌选择方法



3. 结合之前的路径结构体 `Path` 的单点交叉函数，此处对每个个体之间进行交叉，只需要分别调用其内部的成员函数 `singlePointCross()` 即可，同样地变异操作也是如此。

```
void GA::cross(vector<Path>& oth){
    int index = 0;
    int choice_1, choice_2;
    while(index < GROUP_SIZE){
        double pick_pro = ((double)rand()) / RAND_MAX;
        if(pick_pro > CROSS_PRO){
            continue;
        }else {
            // Do groups cross over
            choice_1 = index; choice_2 = index + 1;
            oth[choice_1].singlePointCross(oth[choice_2]);
        }
        index += 2;
    }
}

// Do variation
void GA::varia(vector<Path>& oth){
    int index = 0;
```

```

while(index < GROUP_SIZE){
    double pick_pro = ((double)rand()) / RAND_MAX;
    if(pick_pro < VARIA_PRO){
        oth[index].randomVariation();
    }
    index++;
}
}

```

(2) 可视化程序服务端实现

编程语言：NodeJS

采用框架：Node-GYP

实现过程：

1. 为了使 GA 求解的结果能够实时反应在可视化网页上，此处编写服务端程序，动态定时调用后端 GA 对象的 `getShortestPath()` 方法，传递给前端进行实时渲染。
2. 结合 Restful API 设计思想，将服务端设计为提供接口服务，前端 JS 只需要通过对应 URL Query 进行访问，即可获得当前最优路径节点序列，以及当前路径长度(位于序列最后)。
3. 后端与服务端之间通过 `binding.gyp` 进行绑定，同时 `GA-Export.cpp` 定义了各项函数方法的定向，所依赖的命名空间为 `v8`，头文件 `node.h`。


效果展示：

1. 初始数据



A screenshot of a web browser window at localhost:8088/GA-Data. The page displays a single line of text containing a long sequence of numbers, likely representing the initial data for the GA algorithm.

2. 最终结果



A screenshot of a web browser window at localhost:8088/GA-Data. The page displays a single line of text containing a long sequence of numbers, likely representing the final result of the GA algorithm.

(3) 可视化程序前端实现

编程语言：HTML + CSS + JS

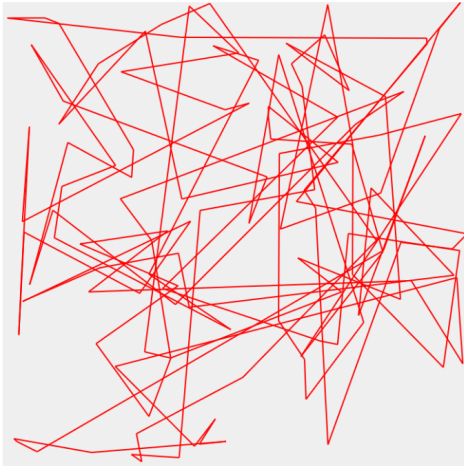
实现过程：

1. 结合 HTML5 绘制图像元素 `canvas`，前端定时对服务端发出请求，该请求使得服务端执行一次迭代(此处定义为长度为50的循环)，并将迭代后的结果返回给前端进行渲染。
2. 当前端连续收到很多个相同的结果时，判断收敛，求解结束。
3. 页面编写结合 Bootstrap 框架，使得各项元素看起来更加美观，同时实时记录当前路径的长度，以及运算总时长，均记录在页面的右侧。

效果展示：

Genertic Algorithm to solve TSP

Start GA



Current Path's Length

23448

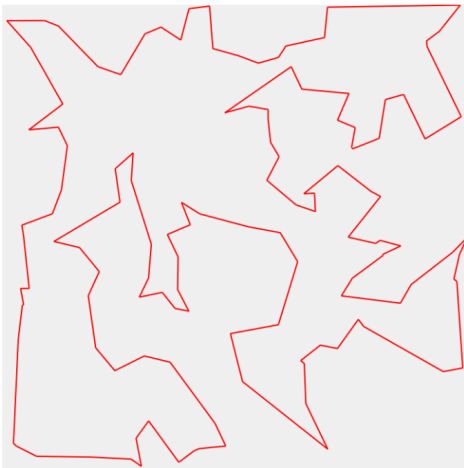
Total Cost Time

0.4s

四、实验结果分析

Genertic Algorithm to solve TSP

Start GA



Current Path's Length

6351

Total Cost Time

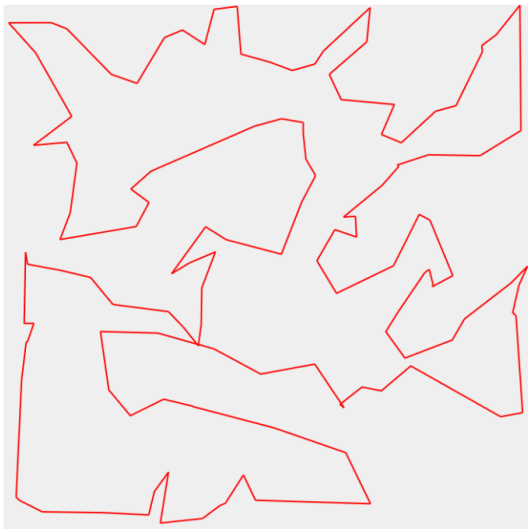
10.9s

此处的输入样本最优解为 6110

1. 通过变换遗传算法的变异操作，引入了模拟退火算法的邻域搜索方法：单点交换，范围倒置，以及循环左移，可以看到最终解可以达到比较好的状态。
 - 优点：相对于只执行单点变异的操作，结果上是有绝对提升的。
 - 缺点：不可否认的是 GA 的整体运行速率还是较为低下，消耗时间长。

2. 与模拟退火算法的结果比较

遗传算法最终解：



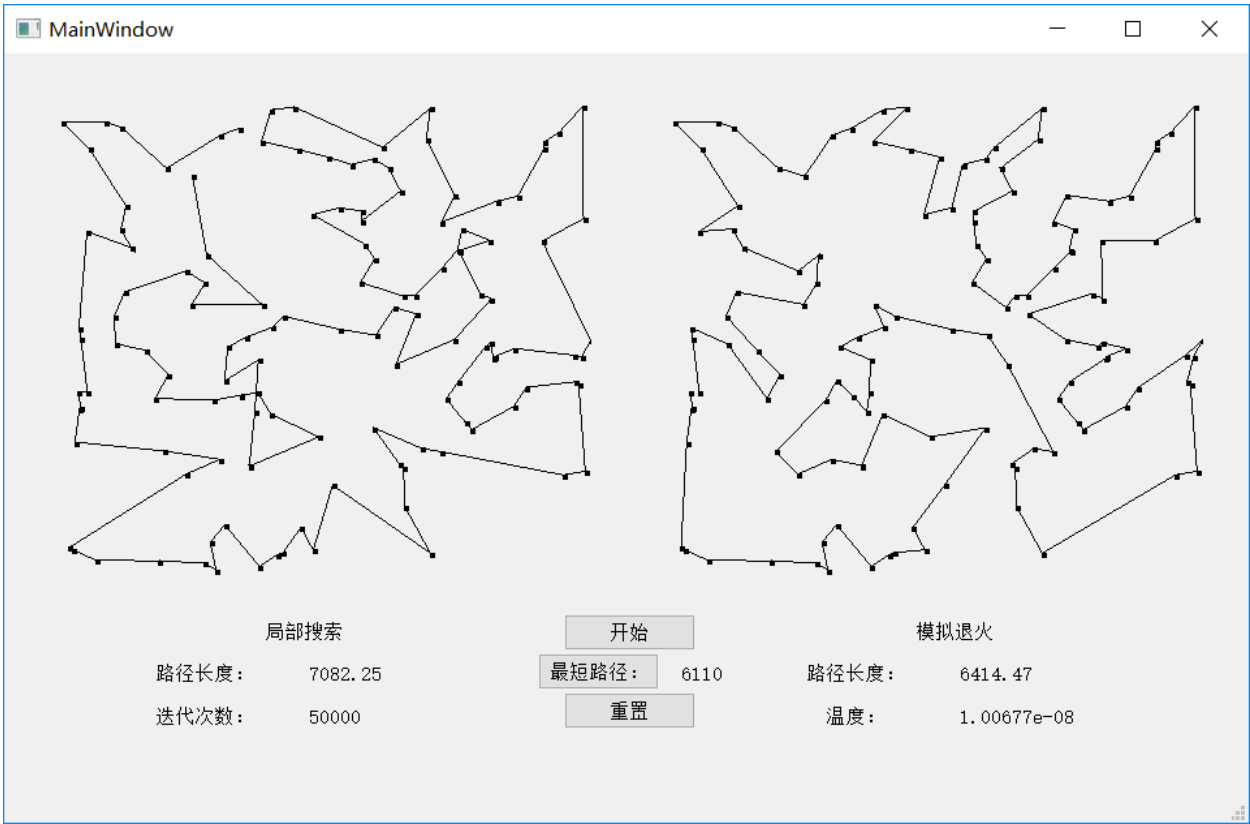
Current Path's Length

6536

Total Cost Time

11.8s

局部搜索与模拟退火算法最终解：



比较分析：

- 总体上来看，此处的遗传算法的最终解相对于模拟退火算法的最终解应该是较差的，同时 SA 能够在短时间内收敛到一个比较好的解，在运行效率上是十分快的。
- 但遗传算法是具有潜力的，适当改变种群内部的交叉、变异操作即可在一定的迭代中获取到较好的解，尽管过程消耗时间较长。

3. 设计高效遗传算法的经验

- 首先各项参数需要设置正确，如变异概率，交叉概率，种群大小等。

- 其次对于交叉操作的选取需要有所权衡，单点交叉虽然实现简单，但是并非就一定低效，个人觉得不同的输入可能对应的最优交叉操作应该有所不同。
- 然后则是保证变异操作的合理性与多样性，能够在一定程度上让父代产生出具有更优性状的子代，同时保留父代优秀个体也是十分重要的。

4. 比较单点交叉与多点交叉的优缺点

- 单点交叉方式是当前使用最多的交叉算法。单点交叉的主要过程是：首先在染色体序列上随机选择一个交换点；然后确定是在交换点前面部分或者后面部分的基因进行交换；最后根据前面的原则将两父本的染色体基因进行交换重组，从而形成了新的个体，即下一代个体。如有两个父本染色体序列 `10010|111` 和 `00101|010`，其中 `|` 表示交换点，按照父本染色体的交换点前部分交换的原则，产生的新得下一代个体的染色体分别是 `00101|111` 和 `10010|010`。
- 多点交叉方式执行过程基本上与单点交叉相同，唯一不同之处在于选取的交换点的个数由一个变为多个，父代个体需要进行的交换行为增多。
- 在本次实验过程中，对路径个体的交叉操作基本采用单点交叉，之前曾尝试过多点交叉(两点交叉操作)，发现整体解的波动范围很大，采用多点交叉的优点在于能够有效跳出局部最优，但这同时也可能使得最终解变成较差解。有时候可以达到 `1~2%` 的较优解，有时却连 `10%` 都无法进入。多点交叉的另一个缺点在于运行时间较于单点交叉要长，由于选取点数的增长，交叉方法执行次数翻倍，故整体运行时间是挺长的。
- 为了将最终解控制在合理的较优解范围内，单点交叉具有运行速度快，同时解范围相对集中的优点，一般测试中都可到达 `3~8%` 的较好成绩，唯一的不足可能就在于其不能有效地跳出局部最优，限制了解上限的提升。

5. 相关的测试视频 `GA.mp4` 已放在测试文件夹 `tc` 中，可以自行查看运行效果。

五、参考文献

1. [遗传算法总结](#)
2. [超详细的遗传算法解析](#)
3. [遗传算法的多种交叉操作介绍](#)