|  |  |  |
| --- | --- | --- |
| 实验报告： | | 遗传算法解决旅行商问题 |
| 学 | 院： | 信息科学与工程学院 |
| 年 | 级： | 2021级硕士 |
| 专 | 业： | 控制科学与工程 |
| 姓 | 名： | 要啥啥不会 |
| 学 | 号： | 只有天知道 |

# 一、具体模型

## 1.旅行商问题:

现有已知的31座城市的坐标位置，要求乘坐飞机周游所有的城市各一次，并最终返回起始城市，求取周游所有城市的最优路径、使得总飞行距离最短。假设：任意两座城市之间的飞行路径为直线，并且不考虑起落距离。（具体的城市坐标数据存储在city0.csv文件）



图1 各城市位置坐标

## 2.具体模型：

本问构建的数学模型可以使用以下形式表达。其中dij表示城市i与城市j之间的距离。其中xij表示是否存在从城市i到城市j的旅程。约束条件表示，每个城市仅可访问一次，即每个城市只可以作为出发城市与到访城市各一次，并且出发城市与到访城市不能为同一城市。





# 二、算法设计

## 1.染色体编码

本文使用换位表达（路径表达或顺序表达）的方式来表示城市之间的访问顺序。染色体中的基因的值表示城市，基因的顺序表示访问城市的顺序。

13,15,6,12,16,7,25,27,3,8,28,29,4,5,10,11,9,1,0,26,30,24,14,23,21,20,17,2,22,18,19

如上编码表示依次到从城市13出发，依次访问城市15、城市6、城市12、城市16、城市7、城市25、城市27、城市3、城市8、城市28、城市29、城市4、城市5、城市10、城市11、城市9、城市1、城市0、城市26、城市30、城市24、城市14、城市23、城市21、城市20、城市17、城市2、城市22、城市18、城市19并最终返回初始城市13。

## 2.交叉算子

本文使用部分映射交叉PMX（Partial-Mapped Crossover）作为程序的交叉算子。PMX由Goldberg和Lingle 提出。可看作二进制串的两点或多点交叉在换位表达中的扩展。用特别的修复程序来解决简单两点交叉引起的非法性。所以PMX是简单的两点交叉和修复程序组合。



图2 部分映射交叉算法示意图

算法示意图如上图所示，具体可以分为以下几个步骤：

1：随机选取两个位置，由两点定义的子串称为映射段；

2：交换双亲的子串，形成原始后代；

3：确定两映射段之间的映射关系；具体为操作为：两个映射段的相同位置的基因互相产生映射。

4：根据映射关系，将后代合法化。具体为操作为：将后代中非映射段的基因与映射段的基因依次进行比较，如果基因相同（即存在重复），更改为该非映射段的基因值为另一个映射段中对应位置的基因，并再次判断，直至该非映射段的基因不在与映射段的基因相同。

### PMX算子程序流程图;



图3 部分映射交叉算法流程图

## 3.变异算子

本文使用反转变异（Inversion Mutation）算子。在染色体上随机选择两个点，并且将两个点之间的子串反转获得变异的染色体。



图4 反转变异算法示意图、流程图

## 4.适值函数

本文通过计算该染色体所表示的路径总长度，来计算适值。适值表示经过所有城市总共走过的路程。总路程越小即始终越小表示个体越优。被筛选留下的可能性也就越大。适值函数使用该染色体通过的总距离表示。



## 5.选择策略

本文采用轮盘赌的筛选策略对种群进行筛选。本文选择将经过交叉、编译算子生成的子代与其父代随机打乱分成10组，每组group\_size个体中选取其中最优的前group\_winner个样本，成为新的一代。其中group\_winner可以使用以下的公式进行计算。





## 6.种群初始化方法

本文采用随机初始化的方法对染色体编码进行初始化。对所有的城市序列随机打乱生成新的种群，该初始化方法存在一定的随机性。

# 三、程序说明

本文使用的程序为c++程序，并且在ubuntu系统中，通过编写CMakeLists.txt调用cmake工具编译实现。没有引用第三方的库，主要使用了STL标准库和cmath进行实现。

## 1.程序中的类

本文主要设计了三个类，用于表示城市位置和计算城市之间距离的类class City{};用于表示种群个体和计算个体的适值的类class Individual{};用于遗传算法迭代计算的类class GA{};。

### 1.1class City{};

类内数据

|  |  |
| --- | --- |
| 城市编号 | int n； |
| 坐标维度 | int dim; |
| 城市位置 | std::vector<double> pose; |
| 与其他城市的距离 | std::vector<double> distance; |

类内函数

|  |  |
| --- | --- |
| 计算城市之间的距离 | Void compute\_distance(std::vector<City> &Citys); |

### 1.2class Individual{};

类内数据

|  |  |
| --- | --- |
| 基因染色体编码 | std::vector<int> genes; |
| 总距离 | double fitness; |
| 适值 | double fitness\_max; |

类内函数

|  |  |
| --- | --- |
| 初始化基因 | Individual(int num); |
| 计算适值 | void evaluate\_fitness(std::vector<City> &Citys); |

### 1.3class GA{};

类内数据

|  |  |
| --- | --- |
| 城市数据 | std::vector<City> Citys; |
| 变异比例 | double mutate\_prob; |
| 种群个体数目 | int Individual\_num; |
| 最大迭代次数 | nt gen\_num; |
| 当前种群 | std::vector<Individual> generation; |
| 当前迭代次数 | int iter = 0; |
| 最优的个体在当前种群的编号 | int best; |

类内函数

|  |  |
| --- | --- |
| 设置遗传算法参数 | GA(std::vector<City> &Citys, int Individual\_num, int gen\_num, double mutate\_prob); |
| 获取当前最优个体 | Individual get\_best(); |
| 迭代训练、保存最优个体 | void train(std::vector<Individual> &best\_generation); |
| 交叉算子 | void PMXcross(); |
| 变异算子 | void Inversionmutate(); |
| 选择下一代 | void PRWselect(); |

## 2.程序中的函数

|  |  |
| --- | --- |
| 随机初始化城市数据 | void init\_citys(std::vector<City> &Citys, int dim); |
| 读取city0.csv文件中城市数据 | void read\_city(std::vector<City> &Citys); |
| 保存城市数据 | void save\_city(std::vector<City> &Citys); |
| 保存实验结果 | void save\_data(std::vector<Individual> &best\_generation, int gen\_num,int indvidual\_num,double mutate\_prob,ofstream &results) |
| 主函数，用于实验 | int main(int argc, char \*\*argv); |

## 3.程序流程图



图5 程序流程图

# 四、仿真结果对比分析

## 1.实验结果路线图



图6 算法求解的路线图

上图是本文使用的遗传算法经过迭代最终获得的实验路线连线图。可以观察出本文所设计使用的遗传算法可以得出较好的近似的最优解，能够有效求解的TSP问题所构建的数学模型。

## 2.迭代次数与收敛性

## 

图7 迭代次数与收敛性实验

上图是种群数量为5000，变异比例为0.1的情况下，使用上述遗传算法迭代计算的适值函数的收敛情况。可见本文使用的算法在迭代次数达到100次时所得的解已经近似是最优解。

## 3.种群数量

图7 改变种群数量实验

上图是种群数量为从50到10000，变异比例为0.1的情况下，使用本文设计遗传算法迭代计算的最佳适值与收敛次数的情况。从中可以得出如下结论：

种群数量越多越是能够保存种群的多样性结果越接近最优值，同时迭代次数也越多，需要也要消耗更多的时间；由于种群存在随机参数，因此每次求解的最优结果也存在一定的随机性，为了获得最接近最优值的解需要多次迭代求取最优解。

## 4.变异比例



图8 改变变异比例实验

上图是种群数量为从5000，变异比例为0.02到0.4直接的情况下，使用本文设计遗传算法迭代计算的最佳适值与收敛次数的情况。从中可以得出如下结论：

变异比例越高，所需要的迭代次数也越高，同时算法的收敛性也会变差，当变异比例过高时算法会变成发散的无法获得近似的最优解。