**实验3 遗传算法求解TSP问题实验**

班级： 学号： 姓名：

**一、实验目的：**

熟悉和掌握遗传算法的原理、流程和编码策略，并利用遗传算法求解组合优化问题，理解求解TSP问题的流程并测试主要参数对结果的影响。

**二、实验原理：**

旅行商问题，即TSP问题（Traveling Salesman Problem）是数学领域中著名问题之一。假设有一个旅行商人要拜访n个城市，他必须选择所要走的路径，路经的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。TSP问题是一个组合优化问题。该问题可以被证明具有NPC计算复杂性。因此，任何能使该问题的求解得以简化的方法，都将受到高度的评价和关注。

遗传算法的基本原理是通过作用于染色体上的基因寻找好的染色体来求解问题，它需要对算法所产生的每个染色体进行评价，并基于适应度值来选择染色体，使适应性好的染色体有更多的繁殖机会，在遗传算法中，通过随机方式产生 若干个所求解问题的数字编码，即染色体，形成初始种群；通过适应度函数给每 个个体一个数值评价，淘汰低适应度的个体，选择高适应度的个体参加遗传操作，经过遗产操作后的个体集合形成下一代新的种群，对这个新的种群进行下一轮的进化。本实验要求利用遗传算法求解TSP问题的最短路径。

**三、实验结果**

1.

表1 遗传算法求解不同规模的TSP问题的结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 城市规模 | 种群规模 | 最大迭代步数 | 独立运行次数 | 最好适应度 | 最差适应度 | 平均适应度 | 平均运行时间 |
| 10 | 100 | 100 | 10 | 25.1652 | 26.5523 | 25.6498 | 54.5s |
| 30 | 100 | 100 | 10 | 11037 | 11962 | 11496.8 | 79.8 |
| 100 | 100 | 100 | 10 | 44670.9 | 47249.8 | 45697.9 | 317.4s |

2.

表2 不同的变异策略和个体选择概率分配策略的求解结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 变异策略 | 个体选择概率分配 | 最大迭代步数 | 独立运行次数 | 最好适应度 | 最差适应度 | 平均适应度 | 平均运行时间 |
| 两点互换 | 按适应度比例分配 | 100 | 10 | 8897.21 | 10147.3 | 9661.37 | 78.7s |
| 两点互换 | 按线性排序 | 100 | 10 | 9347.45 | 10554.5 | 10100.34 | 89.3s |
| 相邻两点互换变异 | 按适应度比例分配 | 100 | 10 | 9834.54 | 10654.5 | 10382.44 | 90.4s |
| 相邻两点互换变异 | 按线性排序 | 100 | 10 | 9933.76 | 11490.1 | 11001.33 | 100.1s |

3.

1. 改进算法策略及核心代码

改进算法策略采用变异操作解决同一问题。算法的核心思路是，在每次迭代中，解的个体随机按4个为1组，每组中只保留最优解，然后对此最优解进行左右翻转、交换、向前移位三种变异操作，生成三个新个体。再参与下次迭代。整个算法不需要计算归一化适应度，核心代码如下：

//初始化种群的方法 加入了轮盘赌的思想

void Population::Initial\_Population(){

//这里的随机函数rand()需要随机数种子，先写在这里，看后面需不要变动

Pc\_ = Pc;

Pm\_ = Pm;

Generation\_ = 0;

std::vector<Individual> Genometemp;

srand((unsigned int) time(NULL));

vecPop\_.clear();

vecBestIndividual\_.clear();

Genometemp.clear();

//vecParentsBabiesRoom\_.clear();

for (int i = 0; i < POPULATION\_SIZE; ++i){//产生种群数目的个体

for (int i = 0; i < POPULATION\_SIZE; ++i) {

Genometemp.push\_back(Individual());//没这句会报vector subscript out of rang

//vecParentsBabiesRoom\_.push\_back(Individual());

//开始的时候直接用的等号赋值，不对的，对向量插入要用push\_back =用于修改

Genometemp[i].Genome\_.push\_back((rand() / double(RAND\_MAX)) \* (GENOME1\_RIGHT - GENOME1\_LEFT) + GENOME1\_LEFT);

Genometemp[i].Genome\_.push\_back((rand() / double(RAND\_MAX)) \* (GENOME2\_RIGHT - GENOME2\_LEFT) + GENOME2\_LEFT);

}

for (int i = 0; i < POPULATION\_SIZE; ++i){

double x1 = Genometemp[i].Genome\_[0];

double x2 = Genometemp[i].Genome\_[1];

Genometemp[i].Fitness\_ = 21.5 + x1 \* sin(4 \* 3.1415926\*x1) + x2 \* sin(20 \* 3.1415926\*x2);

//Genometemp[i].Fitness\_ = 1 + x1\*exp(-x1\*x1 - x2\*x2);

//Genometemp[i].Fitness\_ = 1 + sin(sqrt(x1\*x1 + x2\*x2)) / sqrt(x1\*x1 + x2\*x2);

//Genometemp[i].Fitness\_ = x1\*x1 + x2\*x2;

//Genometemp[i].Fitness\_ = pow(x1\*x1+x2\*x2,0.25)\*(1+sin(50\*pow(x1\*x1 + x2\*x2, 0.1))\*sin(50 \* pow(x1\*x1 + x2\*x2, 0.1)));

//Genometemp[i].Fitness\_ = x1\*x1 - cos(18 \* x1) + x2\*x2 - cos(18 \* x2) + 3;

//Genometemp[i].Fitness\_ = 0.5 - (sin(sqrt(x1\*x1 + x2\*x2))\*sin(sqrt(x1\*x1 + x2\*x2)) - 0.5) / ((1 + 0.001\*(x1\*x1 + x2\*x2))\*(1 + 0.001\*(x1\*x1+ x2\*x2)));

//std::cout << "个体 " << i << " 基因位1： " << x1 << " 基因位2： " << x2 <<" 适应度： "<< vecPop\_[i].Fitness\_ << std::endl;

}

sort(Genometemp.begin(), Genometemp.end());

vecPop\_.push\_back(Genometemp[0]);

Genometemp.clear();

}

}

#if 1

//轮盘法

//选中后新的个体就不会再参与下一轮选择了

void Population::Selection(){

vecParentsBabiesRoom\_.clear();

for (int i = 0; i < (POPULATION\_SIZE / 2); ++i) { //选择一半个体体进行繁衍

Seqencing(vecPop\_);

double sum = 0;

double randomnumber = (rand() / (double)RAND\_MAX)\*totalfitness\_;

std::vector<Individual>::iterator it;

for (it = vecPop\_.begin(); it != vecPop\_.end(); ++ it ) {

sum += (\*(it)).Fitness\_;

if (sum >= randomnumber){

vecParentsBabiesRoom\_.push\_back(\*it);

vecPop\_.erase(it);

break;

}

}

}

}

#endif

//实数编码

#if 1

void Population::crossover(){

Individual childgenome1 = Individual(); //定一个子代染色体的局部变量

Individual childgenome2 = Individual(); //采用一对夫器产两个子代的方法

double a1 = 0, a2 = 0, r1 = 0, r2 = 0;

double genome11 = 0, genome12 = 0, genome21 = 0, genome22 = 0;

std::vector<Individual>::iterator it;

for (int i = 0; i <POPULATION\_SIZE / 4; ++i）{//为什么是八呢?,32个个体 16个被选择做父代，形成8对父母

if ((rand() / (double)RAND\_MAX) <= Pc\_) { //设定了70%的基因交叉概率 交叉方法用的是 适合实数的融合交叉

a1 = rand() / (double)RAND\_MAX;

a2 = rand() / (double)RAND\_MAX;

r1 = rand() / (double)RAND\_MAX;

r2 = rand() / (double)RAND\_MAX;

genome11 = a1\*(vecParentsBabiesRoom\_[2 \* i].Genome\_[0] - vecParentsBabiesRoom\_[2 \* i + 1].Genome\_[0]) + r1\*vecParentsBabiesRoom\_[2 \* i].Genome\_[0] + (1 - r1)\*vecParentsBabiesRoom\_[2 \* i + 1].Genome\_[0];

genome21 = a1\*(vecParentsBabiesRoom\_[2 \* i].Genome\_[0] - vecParentsBabiesRoom\_[2 \* i + 1].Genome\_[0]) + r1\*vecParentsBabiesRoom\_[2 \* i + 1].Genome\_[0] + (1 - r1)\*vecParentsBabiesRoom\_[2 \* i].Genome\_[0];

genome12 = a2\*(vecParentsBabiesRoom\_[2 \* i].Genome\_[1] - vecParentsBabiesRoom\_[2 \* i + 1].Genome\_[1]) + r2\*vecParentsBabiesRoom\_[2 \* i].Genome\_[1] + (1 - r2)\*vecParentsBabiesRoom\_[2 \* i + 1].Genome\_[1];

genome22 = a2\*(vecParentsBabiesRoom\_[2 \* i].Genome\_[1] - vecParentsBabiesRoom\_[2 \* i + 1].Genome\_[1]) + r2\*vecParentsBabiesRoom\_[2 \* i + 1].Genome\_[1] + (1 - r2)\*vecParentsBabiesRoom\_[2 \* i].Genome\_[1];

if (genome11 > 12.1 || genome11 < (-3.0) || genome21 > 12.1 || genome21 < (-3.0)){

genome11 = r1\*vecParentsBabiesRoom\_[2 \* i].Genome\_[0] + (1 - r1)\*vecParentsBabiesRoom\_[2 \* i + 1].Genome\_[0];

genome21 = r1\*vecParentsBabiesRoom\_[2 \* i + 1].Genome\_[0] + (1 - r1)\*vecParentsBabiesRoom\_[2 \* i].Genome\_[0];

}

if (genome12 > 5.8 || genome12 < 4.1 || genome22 > 5.8 || genome22 < 4.1){

genome12 = r2\*vecParentsBabiesRoom\_[2 \* i].Genome\_[1] + (1 - r2)\*vecParentsBabiesRoom\_[2 \* i + 1].Genome\_[1];

genome22 = r2\*vecParentsBabiesRoom\_[2 \* i + 1].Genome\_[1] + (1 - r2)\*vecParentsBabiesRoom\_[2 \* i].Genome\_[1];

}

childgenome1.Genome\_.push\_back(genome11);

childgenome1.Genome\_.push\_back(genome12);

childgenome2.Genome\_.push\_back(genome21);

childgenome2.Genome\_.push\_back(genome22);

vecParentsBabiesRoom\_.push\_back(childgenome1);

vecParentsBabiesRoom\_.push\_back(childgenome2);

childgenome2.Genome\_.clear();

childgenome1.Genome\_.clear();

}

else {//不交叉的时候，子代由父代基因偏离

vecParentsBabiesRoom\_.push\_back(vecParentsBabiesRoom\_[2 \* i]);

vecParentsBabiesRoom\_.push\_back(vecParentsBabiesRoom\_[2 \* i + 1]);

}

}

}

#endif

1. 改进算法的主要参数设置

种群规模：50、100、200、500、1000、2000

交叉概率：0.8 变异概率：0.8 最大迭代步数：100

1. 改进算法最后求得的34 个省会的最短路径值、最优个体和算法运行时间

|  |  |
| --- | --- |
| 种群规模 | 最短路径值 |
| 50 | 1304.7587 |
| 100 | 1302.0629 |
| 200 | 1298.1412 |
| 500 | 1295.7248 |
| 1000 | 1295.7248 |
| 2000 | 1299.9698 |
| 最优个体 | 1295.7248 |
| 算法运行时间 | 637.4s |

1. 在相同的参数设置下，用基本遗传算法（没有使用改进策略）求得的34 个省会的最短路径值、最优个体和算法运行时间

|  |  |
| --- | --- |
| 种群规模 | 最短路径值 |
| 50 | 2679.7 |
| 100 | 2053.0 |
| 200 | 1747.6 |
| 500 | 1489.3 |
| 1000 | 1499.8 |
| 2000 | 1624.6 |
| 最优个体 | 1489.3 |
| 算法运行时间 | 518.7s |

**四、实验总结**

1. 完成实验报告要求1, 2和3。

1、分析遗传算法求解不同规模的TSP问题的算法性能。

规模越大，算法的性能越差，所用时间越长。

1. 增加1种变异策略和1种个体选择概率分配策略，比较求解30个城市的TSP问题时不同变异策略及不同个体选择分配策略对算法结果的影响。

不同变异策略和不同个体选择分配策略几乎不影响算法运行时间，但会影响适应度。

1. 比较分析遗传算法的改进策略对求解34个城市的TSP问题的结果的影响。

遗传算法的改进策略比基本遗传算法所得结果更优化，从寻优历史可见，迭代次数在接近1000次才能得到近似最优解，总体而言，改进遗传算法收敛性更好，但计算量稍大。

2. 实验心得体会

通过本实验，我更加深入体会了参数设置对算法结果的影响。同一个算法，参数值不同，获得的结果可能会完全不同。

同时，通过本次实验的实践，我对遗传算法有了更进一步的了解。遗传算法是一种智能优化算法，它能较好地近似求解TSP问题，在问题规模较大时，遗传算法的优势就明显体现出来，当然不能完全保证能得到最优解。

遗传算法的实现有些关键点，一是串的编码方式，本质就是问题编码，串长度及编码形式对算法收敛影响极大；二是适应函数的确定，这是选择的基础，应具休问题具休分析，沒有大一统的方法；三是自身参数的设定，其中重要的是种群规模，最大迭代次数，交叉概率和变异概率，通过实验我们可以看到种群规模、最大迭代次数对问题求解的精度有很大影响，交叉概率和变异概率的设定对间题的收敛速度和求解精度也都有极大的影响，但在不同条件下，这种影响表现的程度有很大的不同。具体参数的应根据具休的领域问题设定。