昆明理工大学

课程报告

课程名称：人工智能(0403909)

学　　院：信息工程与自动化学院

专业年级：2020 级

学　　号：202010401137

学生姓名：丁紫嫣

指导教师：钱谦

日　　期：2022-5-31

目录

一、课程简介

1、综述

人工智能作为计算机科学的一个重要分支，是一门理论基础完善、多学科交叉且应用领域广阔的前沿学科，主要研究如何利用计算机模拟、延伸和扩展人类的智能活动，即通过研究如何使计算机更聪明、能运用知识处理问题、可模拟人类的智能行为，进而揭示人类智能的根本机理。其主要任务是建立或运用智能信息处理理论，设计并实现可展现某些近似于人类智能行为的计算机系统。

2、主要教学内容

课程目的是使学生在已有计算机知识基础上，对人工智能从整体上形成较全面和清晰的系统认识，掌握人工智能的基本概念、基本原理和基本方法，了解人工智能研究与应用的新进展和新方向，开阔学生知识视野、提高解决问题的能力，为将来更加深入的学习和运用人工智能相关理论和方法解决实际问题奠定初步基础。

为此，课程选择人工智能领域中一些具有代表性的内容进行重点介绍。首先对人工智能的起源与发展，以及人工智能领域影响较大的主要流派及其认知观进行简要概述；然后介绍人工智能中的几种经典技术，如推理证明技术、问题求解技术等；此外，还对当前人工智能最热门的研究和应用领域，如计算智能等技术进行讨论。具体内容包括：

2.1 人工智能的起源与发展

人工智能自 1956 年诞生以来，其发展并非一帆风顺，从激烈争论到今天的可喜局面，其过程可谓艰辛。本部分以人工智能的有关概念及定义为切入点，通过分析其异同，从不同角度介绍人工智能的起源与发展，使学生对人工智能这一学科形成较全面的认识。

2.2 人工智能的主要流派

从符号主义为代表的经典人工智能到连接主义、行为主义，人工智能的研究从一家独秀走向百家争鸣。本部分针对人工智能的主要流派，通过对各流派的理论基础及其认知观介绍，使学生对人工智能的研究方向及应用领域形成较深入的了解。

2.3 经典人工智能的推理技术

经典人工智能的有关推理技术和方法，对人工智能学科的发展产生了极其深远的影响，是认识和了解人工智能研究领域的重要途径，也是初步掌握人工智能相关技术方法的主要手段。本部分主要介绍基于数理逻辑方法的推理证明技术，尤其是消解原理这一定理证明方法的典型代表。

2.4 问题求解与搜索技术

问题求解技术作为人工智能研究领域的一个核心问题，其主要涉及知识表示和求解搜索两个方面。本部分将主要介绍问题求解中的几种常用方法，如状态空间、问题规约、与或图、产生式系统、谓词逻辑、语义网络、知识图谱等，并在图搜索方法与求解策略分析的基础上，逐步引入启发式搜索方法。

2.5 计算智能

尽管经典方法是初步形成人工智能研究与应用能力的重要途径，但随着技术的发展与应用的延伸，当前以机器学习和人工神经网络为代表的计算智能已成为研究热点，并发展成为智能学科中新的增长点，本部分即针对机器学习和人工神经网络中的基础知识作概述性介绍。

3、课程教学目标

3.1 知识与技能目标

了解人工智能的发展与研究内容，掌握人工智能的基本概念、基本思想方法和重

要算法，熟悉典型的人工智能系统，学习用启发式搜索求解问题，了解有关机器学习与人工神经网络的基本原理，初步具备用经典人工智能方法解决一些简单实际问题的能力。

3.2 过程与方法目标

通过学习掌握人工智能的基本概念、基本思想方法和重要算法，了解人工智能研究与应用的新进展和新方向，拓展学生知识视野，为进一步学习和运用人工智能相关理论方法解决实际问题奠定初步基础。

3.3 科学研究发展目标

通过课程学习，对人工智能从整体上形成较清晰全面的了解，更重要的是培养学生积极思考、严谨创新的科学态度和解决实际问题的能力。

二、上机题简介

用 C++语言编写和调试一个基于**宽度优先搜索**法的解决“野人与传教士过河”问题的程序以及一个用遗传算法解旅行商 TSP 问题的程序。目的是学会运用知识表示方法和搜索策略求解一些考验智力的简单问题，熟悉简单智能算法的开发过程并理解其实现原理。

题目一：野人与传教士渡河问题：3 个野人与 3 个传教士打算乘一条船到对岸去，该船一次最多能运 2 个人，在任何时候野人人数超过传教士人数，野人就会把传教士吃掉，如何用这条船把所有人安全的送到对岸？在实现基本程序的基础上实现 N 个野人与 N 个传教士问题的所有解的求解，N 由用户输入。

题目二：用遗传算法解旅行商 TSP 问题：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，路径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。

三、报告内容（重点）

通过解决野人与传教士问题，学习了盲目搜索中的宽度优先搜索和广度优先搜索，以及启发式搜索中的 A*算法。

1. 基于宽度优先搜索（BFS）“331 野人与传教士渡河”问题 C++ 求解

1.1 问题的分析

野人与传教士渡河问题：3 个野人与 3 个传教士打算乘一条船到对岸去，该船一次最多能运 2 个人，在任何时候野人人数超过传教士人数，野人就会把传教士吃掉，如何用这条船把所有人安全的送到对岸？在实现基本程序的基础上实现 N 个野人与 N 个传教士问题的所有解的求解，N 由用户输入。

首先对问题作如下抽象，以列表 state=[a,b,c]分别代表初始岸边的传教士人数，野人人数，船只数目，有：

初始状态：state=[3,3,1]

agent 所有可能行动：

当 c==1 时,在以下五种状态中选择一种执行：（运输执行顺序：1.两个传教士 2.两个野人 3.一个野人一个传教士 4.一个传教士 5.一个野人）

注意，把 c=1 变成 c=0

a=a-2, c=0

b=b-2, c=0

b=b-1, a=a-1, c=0

a=a-1, c=0

b=b-1, c=0

但是需保证执行动作 state 处于状态空间之中，否则不能执行。

当 c==0 时,在以下五种状态中选择一种执行：（执行顺序：1.两个传教士 2.两个野人 3.一个野人一个传教士 4.一个传教士 5.一个野人）

注意，把 c=0 变成 c=1

a=a+2, c=1

b=b+2, c=1

b=b+1, a=a+1, c=1

a=a+1, c=1

b=b+1, c=1

但是需保证执行动作 state 处于状态空间之中，否则不能执行。

状态空间：

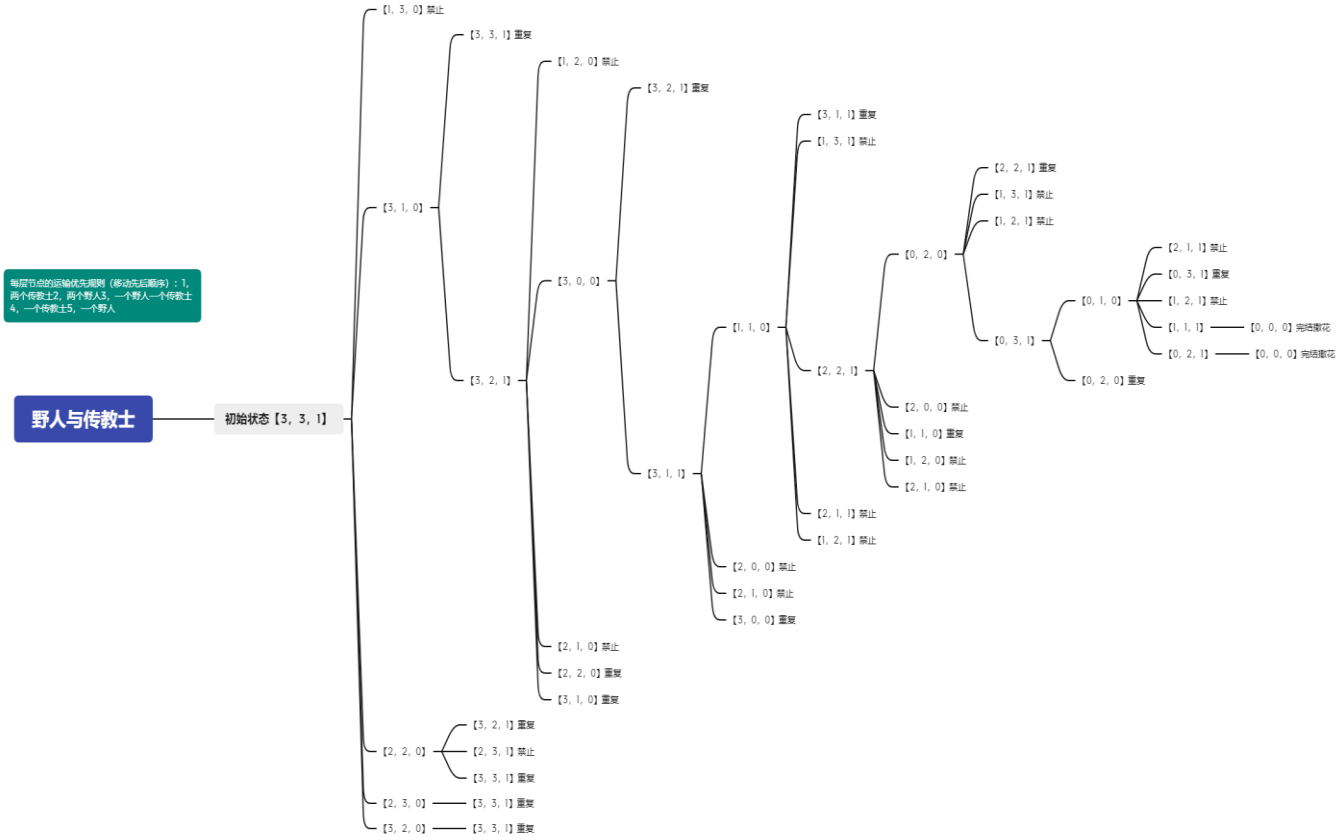
[3, 3, 1], [3, 2, 1], [3, 1, 1], [3, 0, 1], [2, 2, 1], [1, 1, 1], [0, 3, 1], [0, 2, 1], [0, 1,

0]

[0, 0, 0], [3, 2, 0], [3, 1, 0], [3, 0, 0], [2, 2, 0], [1, 1, 0], [0, 3, 0], [0, 2, 0], [0, 1, 0]

因此我们用 xmind 绘制了一张状态空间图，如下：

（为了有序处理，我们设定了每层节点生成的运输优先规则（移动人口的先后顺序）：1.两个传教士 2.两个野人 3.一个野人一个传教士 4.一个传教士 5.一个野人）



宽度优先搜索：

BFS，其英文全称是 Breadth First Search。 BFS 并不使用经验法则算法，属于一种盲目搜寻法，目的是系统地展开并检查图中的所有节点，以找寻结果。换句话说，它并不考虑结果的可能位置，彻底地搜索整张图，直到找到结果为止。从算法的观点，所有因为展开节点而得到的子节点都会被加进一个先进先出的队列中。一般的实验里，其邻居节点尚未被检验过的节点会被放置在一个被称为 open 的容器中（例如队列或是链表），而被检验过的节点则被放置在被称为 closed

的容器中。（open-closed 表）

显然，宽度优先搜索是按层查找的，也就是说以接近起始节点的程度依此拓展节点，搜索逐层进行，在对下一层的任一节点搜索前，必须搜索完本层的所有节点。算法描述如下：

（1） 把起始节点放入 open 表中（如果该起始节点为一目标节点，则求得一个解答）。

（2） 如果 open 表是一个空表，则没有解，失败退出，否则继续。

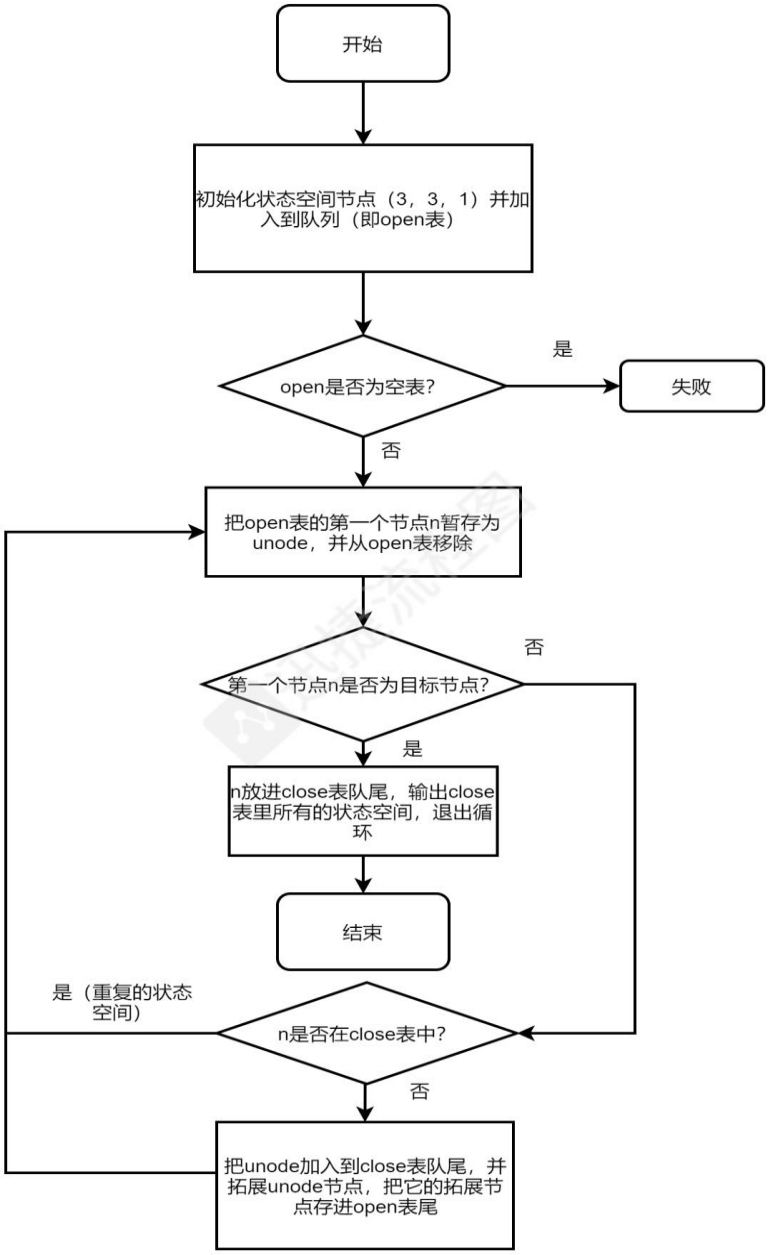
（3） 把第一个节点（节点 n）从 open 表中移出，并把它放入 closed 表的拓展节点表中。

（4） 拓展节点 n,如果没有后继节点，则转向上述步骤（2）。

（5） 把 n 的所有后继节点放到 open 表的末端，并提供这些后继节点回到 n 的指针。

（6） 如果 n 的任一个后继节点是个目标节点，则找到一个解答，成功退出，否则转向步骤（2）。

使用宽度优先算法的野人与传教士渡河问题表示成程序框图如下：



1.2 主要数据结构

主要结构体：

MCNode 表示节点，也就是一个状态空间：

```
typedef struct
{
    int m;//表示传教士
    int c;//  表示野人
```

int b;//船状态,1 为此岸，0 为不在起始子岸

}MCNode;

Open 表存储已生成而未考察的节点，Close 表记录已访问过的节点：

list<MCNode> open;//相当于队列

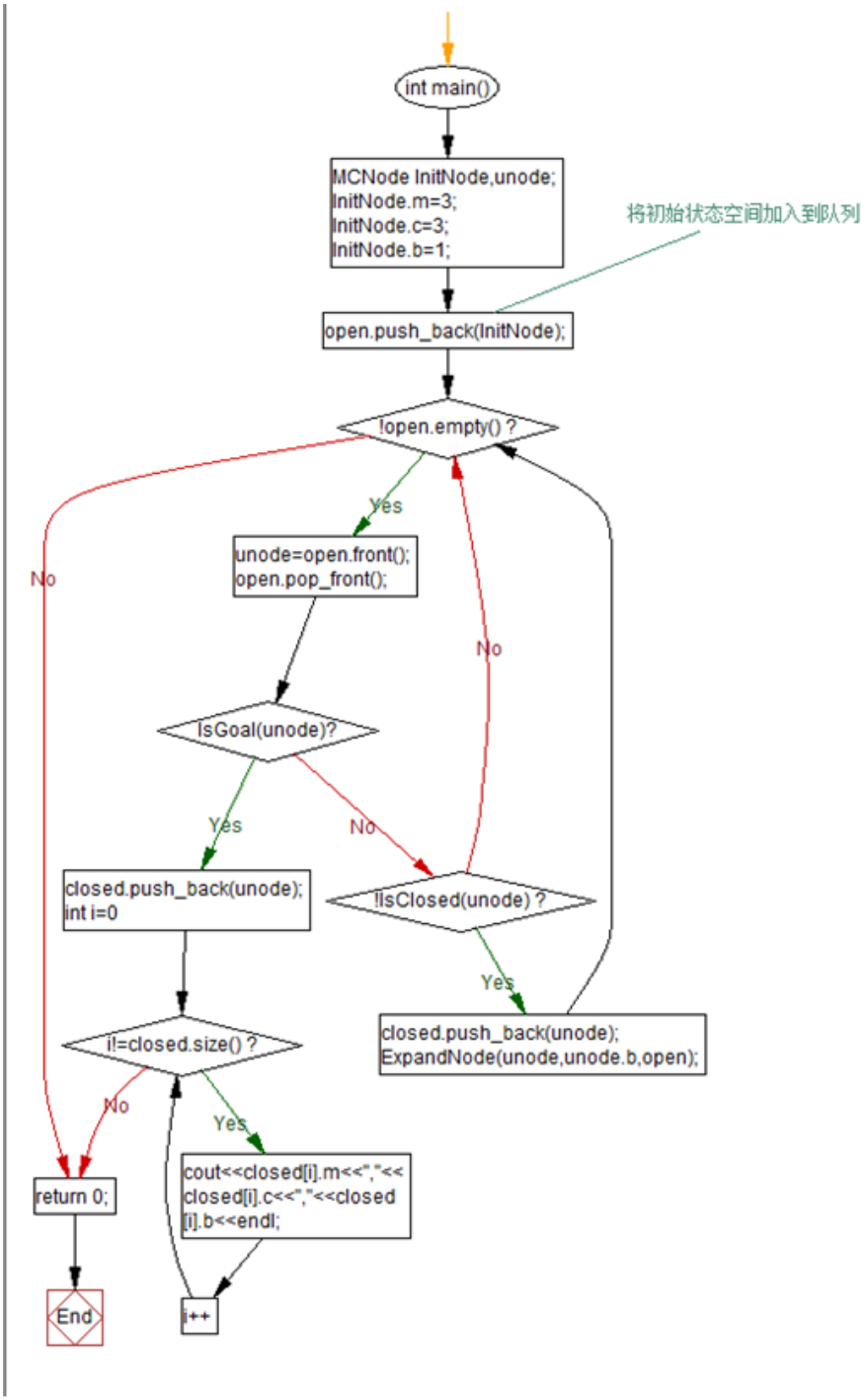
vector<MCNode> closed;//closed 表

主要函数：

```
bool IsGoal(MCNode tNode)   //判断是否目标节点（0，0，0）
bool IsLegal(MCNode tNode)   //判断节点是否合法
bool operator==(MCNode m1,MCNode m2) //判断节点是否重复
bool IsClosed(MCNode tNode)  //判断节点是否已经遍历过（存在 closed 表中）？
void ExpandNode(MCNode tNode,int b,list<MCNode> &open)   //拓展节点函数，运用 5 条规则进
```

行拓展

main 函数的结构框图如下：

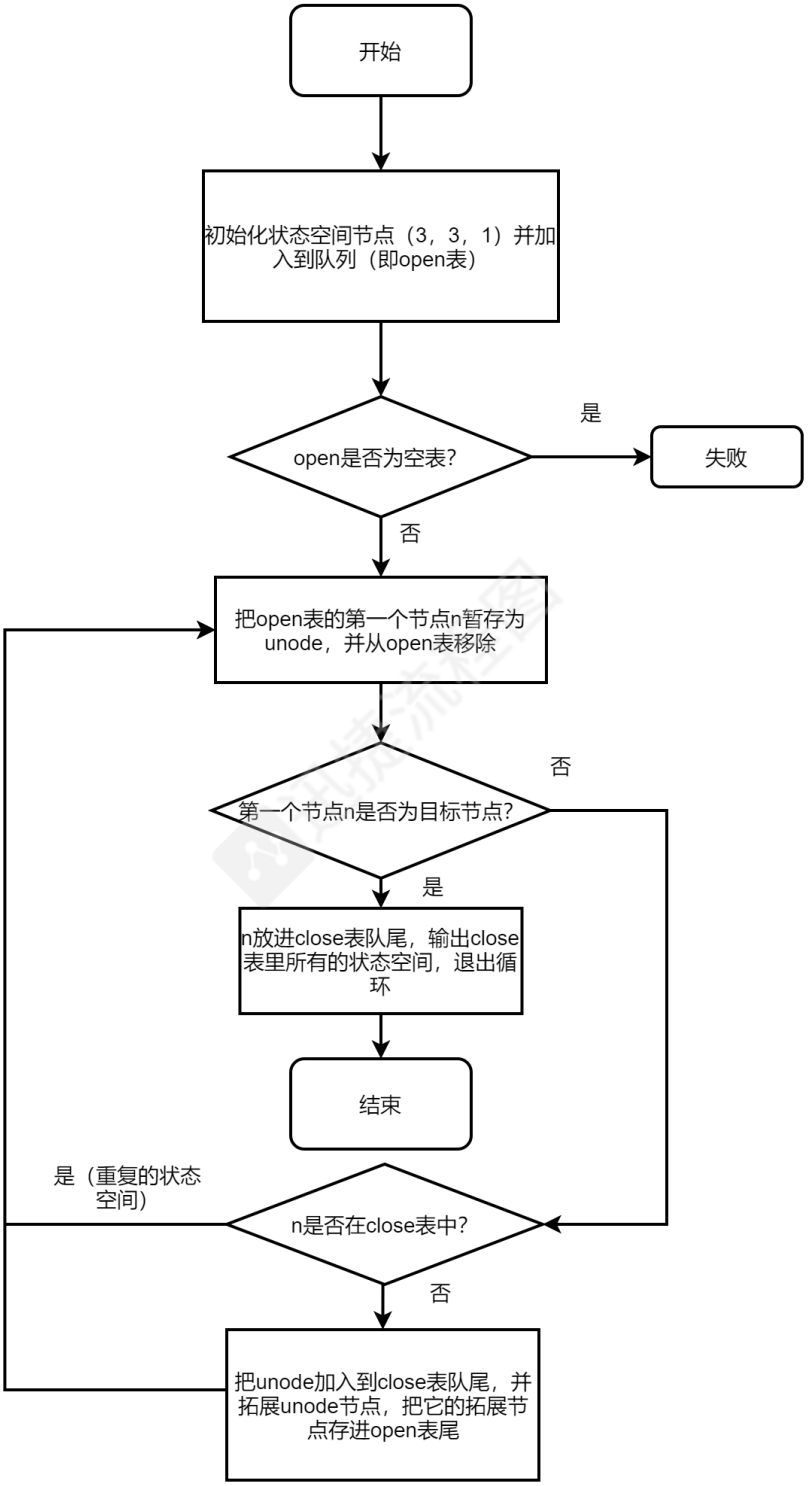


2. 基于深度优先搜索（DFS）“野人与传教士渡河”问题 C++ 求解

事实上，深度优先搜索与宽度优先搜索一样都属于盲目搜索。在深度优先搜索中，首先拓展的是最新生成的（即最深的）节点，深度相等的节点可以任意排列。放到 331 野人与传教士渡河问题上来，即每访问节点的顺序不必再按照我们规定的 1.两个传教士 2.两个野人 3.一个野人一个传教士 4.一个传教

士 5.一个野人，如果想达到深度优先搜索，把一条路径生成的子节点访问完再访问下一条路径，只需要把广度优先中对 open 表的访问由先进后出改为先进先出。

流程图如下：



可以说，两种方法差距并不大，程序上只有一条代码的区别：

A.　　open.push_front(node[i]);//队列后进先出，深度优先搜索，最后得到一条最小解序列

B. open.push_back(node[i]);//队列后进后出，广度优先搜索，最后得到最小解序列状态空间图

3. 基于宽度优先搜索（BFS）“NNk 野人与传教士渡河”问题 C++ 求解

3.1 问题的分析与求解

当有 N 个野人和 N 个传教士，船不止能承载两人：我们要求野人数目等于传教士数目，且船载人数可以自行输入拟定。我们不打算输出最小解（最优解），而是**所有路径**。

我们无法如上面三个野人三个传教士每次最多运两人那样列举规定运输执行顺序：1.两个传教士 2.两个野人 3.一个野人一个传教士 4.一个传教士 5.一个野人，**因为运输规则将不止五条，我们只有用 for 循环嵌套规定运输顺序。**

而且为了让结果更加直观，建议使用队列存储路径，以便清晰呈现每一种解法。这面临一种问题，关于去重复的机制如何拟定。如果按照上文的**三个野人三个传教士解法**，只能输出最小解序列的状态空间图，而不能输出所有解。那么为了设定合适的去重复机制，我们以状态空间图结点的深度为度量值，如果读到该状态空间曾经出现过，且当前它的深度比之前还要深，就代表重复，应该去除。

3.2 主要数据结构

于是，存放状态空间的结构体就长这样：

```
struct States
{
    int X1;//起始岸上的牧师人数
    int X2;//起始岸上的野人人数
    int X3;//小船现在位置(1 表示起始岸，0 表示目的岸)
    int tree_depth = 0;//该状态第一次出现的最低层
    States * pre;//指向前驱的指针
};
```

主要变量有：

```
States * hasgone[1000];//已经走过的状态,用于避免以后重复
int num = 0;//hasgone 的数量
int n = 0;//牧师和野人的总人数
int c = 0;//小船上可承载人数
queue<States *> que;//队列，用于进行广度优先搜索
//保存路径，以及它对应的船来回次数
States *answer[1000];
```

```
int num_answer = 0;
int step_num[1000];//路径对应的步数

int tree_depth = 0;
queue<States*> rubbish;//垃圾队列,用来存放最后运算完未 delete 的数据
```

主要函数：

1.

```
States* Start2Destination(States* pre,int x1, int x2)//now 当前状态,从开始到目的地(x1 是船上牧师人数，x2 是船上野人人数)
{
    int Start_Priests= pre->X1 - x1;//现在岸上的牧师人数
    int Start_savages= pre->X2 - x2;//现在岸上的野人人数
    int End_Priests = n-pre->X1 + x1;//现在目的地的牧师人数
    int End_savages = n-pre->X2 +x2;//现在目的地的野人人数
    if ((x1 >= x2||x1==0)&& (Start_Priests >= Start_savages|| Start_Priests==0) &&( End_Priests >= End_savages|| End_Priests==0))//符合要求
    {
        States * now = new States;
        now->pre = pre;
        now->X1 = Start_Priests;
        now->X2 = Start_savages;
        now->tree_depth = pre->tree_depth + 1;
        now->X3 = 0;
        return now;
    }
    return NULL;
}
```

2.

```
States* Destination2Start(States* pre,int x1, int x2)//从目的地到开始地(x1 是船上牧师人数，x2 是船上野人人数)
{
    int Start_Priests = pre->X1 +x1;//现在岸上的牧师人数
    int Start_savages = pre->X2 + x2;//现在岸上的野人人数
    int End_Priests = n - pre->X1 - x1;//现在目的地的牧师人数
    int End_savages = n - pre->X2 - x2;//现在目的地的野人人数
    if ((x1 >= x2 || x1 == 0) && (Start_Priests >= Start_savages || Start_Priests == 0) && (End_Priests >= End_savages || End_Priests == 0))//符合要求
    {
        States * now = new States;
        now->pre = pre;
        now->X1 = Start_Priests;
        now->X2 = Start_savages;
        now->tree_depth = pre->tree_depth + 1;
```

```
            now->X3 = 1;
            return now;

        }
        return NULL;
    }
```

3. void showSolution(States * states, int time) //展示输出结果函数

```
    {
        if (states == NULL)
        {
            step_num[num_answer] = time;
            cout << endl;
        }
        else
        {
            showSolution(states->pre, ++time);
            cout << "(" << states->X1 << "," << states->X2 << "," << states->X3 << ")
";
        }
    }
```

4. bool Hasgone(States* one)//判断这个是否前面已经走过的去重复函数

```
    {
        for (int i = 0; i < num; i++)
        {
            if (one->X1 == hasgone[i]->X1&&one->X2 == hasgone[i]->X2&&one->X3
==  hasgone[i]->X3&&one->tree_depth>hasgone[i]->tree_depth)//有重复的,且不再同一
层,删除 new 的东西,return,true
            {
                delete one;

                return true;
            }
        }
        //该状态之前未走过
        //加入 hasgone 里面并 num++
        hasgone[num] = one;
        num++;
        return false;
    }
```

5. void Nextstep() //处理出入队列逻辑的函数

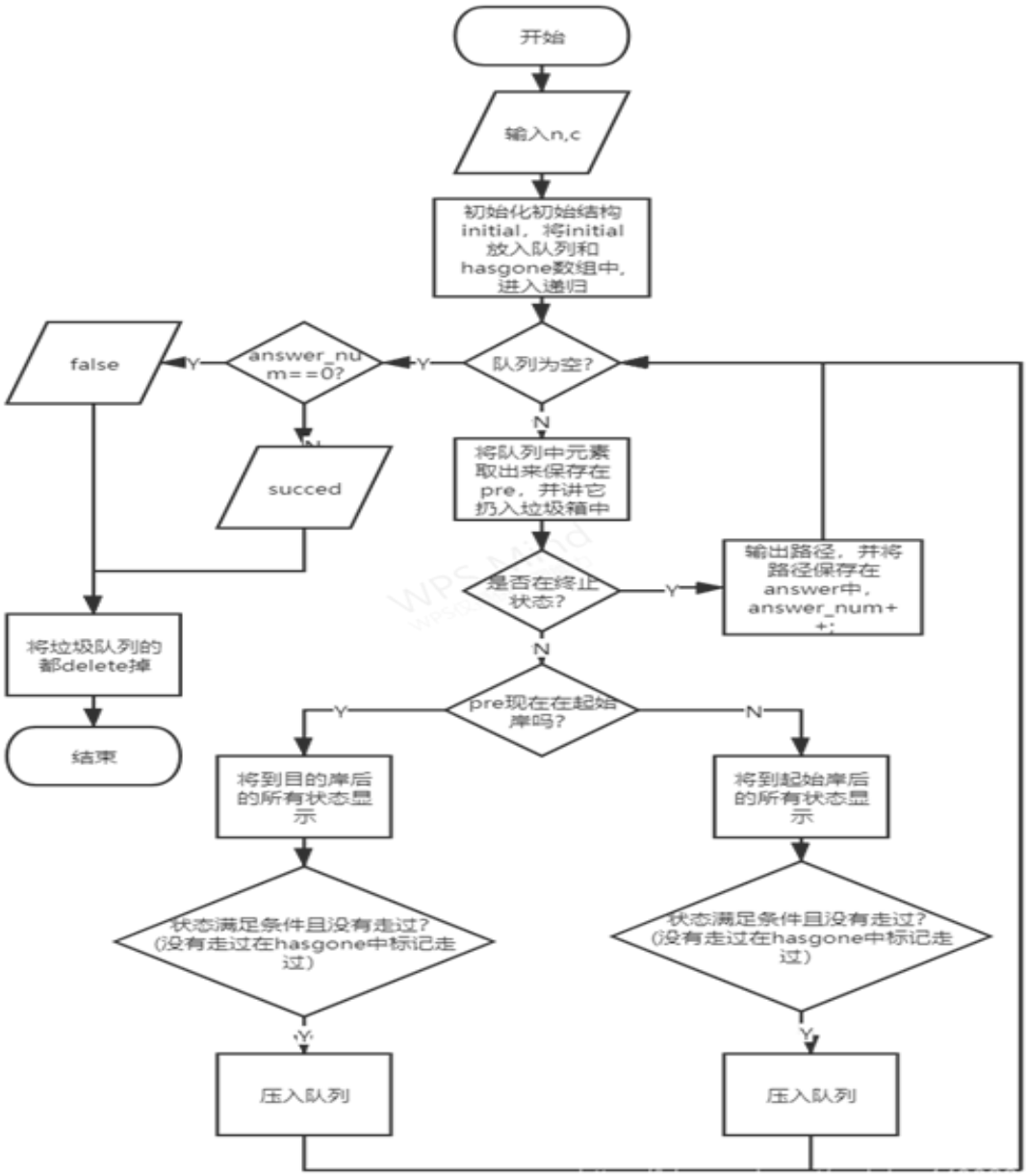
```
{
    if (que.empty()) return;//队列为空
    States*pre = que.front();
    rubbish.push(pre);
    que.pop();
    if (pre->X1 == 0 && pre->X2 == 0 && pre->X3 == 0)//路程已经走完了
    {
        route++;
        cout << "第"<<route<<"条路径：";
        showSolution(pre,0);
        cout << "                路径长度为" << step_num[num_answer] << endl;
        answer[num_answer] = pre;
        num_answer++;
        Nextstep();
    }
    //if (Hasgone(pre)) return;//这个状态之前走过了,返回
    if (pre->X3 == 1)//在起始岸
    {
        int Max1 = pre->X1 > c ? c : pre->X1;
        int Max2 = pre->X2 > c ? c : pre->X2;
        for (int i = 0; i <= Max1; i++)
        {
            for (int j = 0; j <= Max2; j++)
            {
                if (i + j <= c&&i+j>=1)
                {
                    States* now=Start2Destination(pre,i,j);
                    if (now!=NULL&&!Hasgone(now))
                    {
                        que.push(now);
                    }
                }
            }
        }
        Nextstep();
    }
    else//在目的岸
    {
        int Max1 = n - pre->X1 > c ? c : n-pre->X1;
        int Max2 = n - pre->X2 > c ? c : n-pre->X2;
        for (int i = 0; i <= Max1; i++)
```

```
        {
            for (int j = 0; j <= Max2; j++)
            {
                if (i + j <= c && i + j >= 1)
                {
                    States* now = Destination2Start(pre, i, j);
                    if (now != NULL && !Hasgone(now))
                    {
                        que.push(now);
                    }
                }
            }
        }
        Nextstep();
    }
}
```

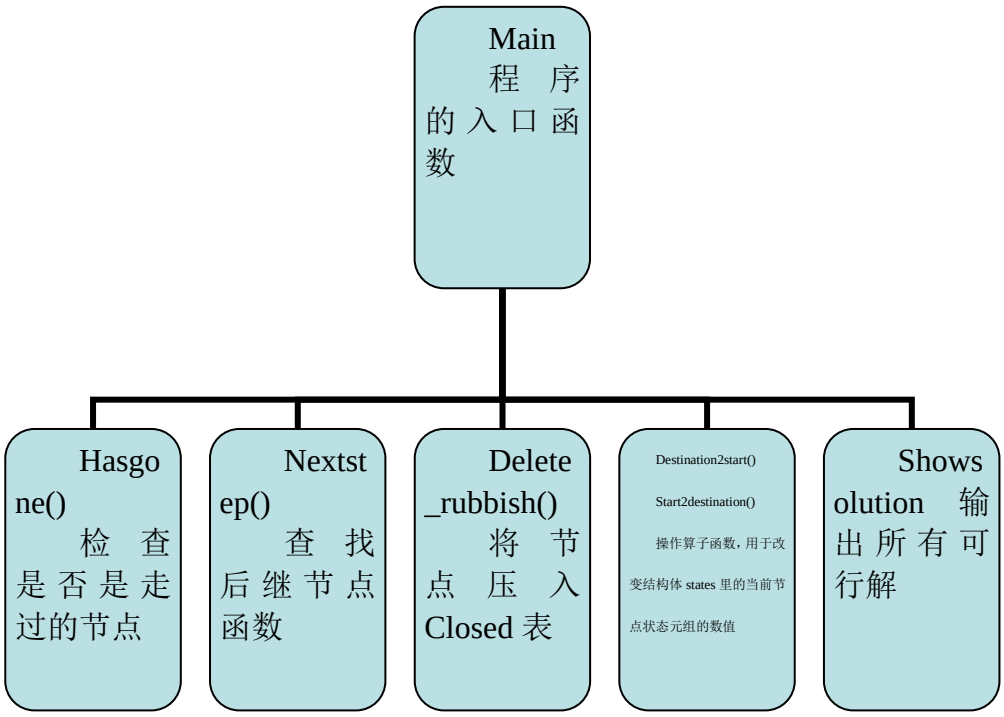
6. void delete_rubbish()//逐步清除垃圾列表

```
{
    while (!rubbish.empty())
    {
        States*del = rubbish.front();
        rubbish.pop();
        delete del;
    }
}
```

程序设计流程图：

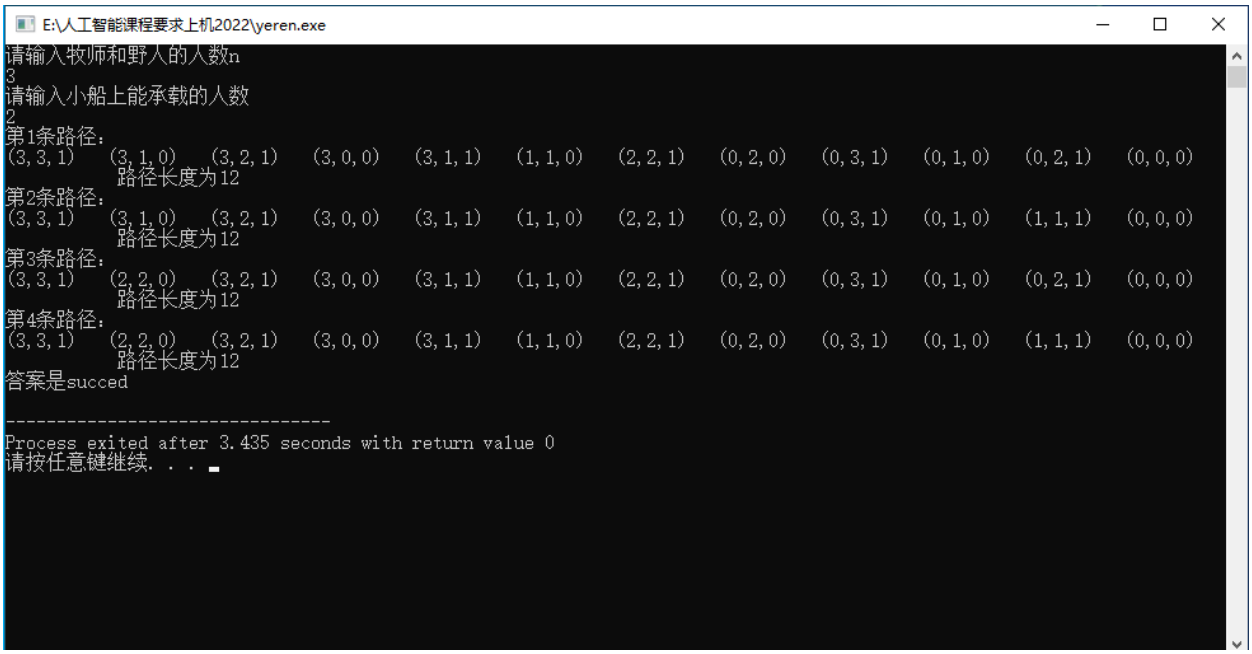

开始
输入n,c
初始化初始结构initial，将initial放入队列和hasgone数组中，进入递归
队列为空?
Y
answer_num==0?
Y
false
succed
N
将队列中元素取出来保存在pre，并讲它扔入垃圾箱中
是否在终止状态?
Y
输出路径，并将路径保存在answer中，answer_num++;
N
pre现在在起始岸吗?
Y
N
将到目的岸后的所有状态显示
将到起始岸后的所有状态显示
状态满足条件且没有走过?(没有走过在hasgone中标记走过)
状态满足条件且没有走过?(没有走过在hasgone中标记走过)
Y
Y
压入队列
压入队列
将垃圾队列的都delete掉
结束

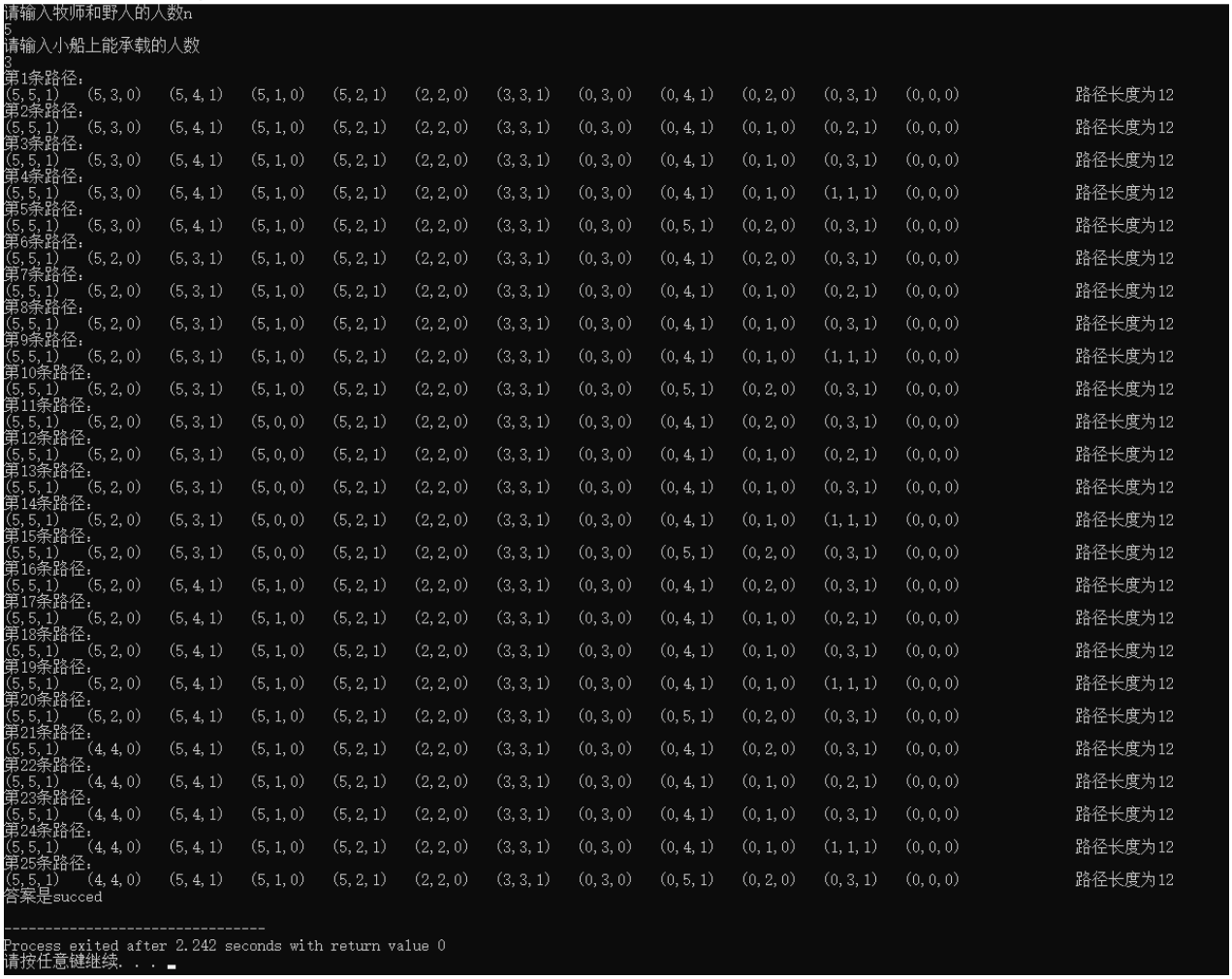
程序功能流程图：



求所有解和求路径最优解的去重复机制有很大不同，即如果求最优解，之前出现过的节点（状态空间）只要再出现就是重复，无论是否处于同一层，就该被去重；而求所有解需要把同一层哪怕相同的节点保留，即去重的限制条件为：如果当前遍历到的结点的深度比它上次出现时记录下的深度要深，就算重复，应当被去重。

程序运行结果：





事实上不是所有情况都有解，比如传教士野人各 7 人渡船每次装载 3 人的 7-7-3 方案便无解。

4. A*算法求 NNk 问题的最优解

盲目搜索容易造成资源浪费，如果要求最优解，我们想到用启发式搜索，A*算法：

用一个三元组来表示左岸状态，即 S=(M,C,B),将所有扩展的节点和原始节点存放在同一列表中。初始状态为（N,N,1），目标状态为（0,0,0），问题的求解转换为在状态空间中，找到一条从状态(N,N,1)到状态(0,0,0)的最优路径。

例：在 3 传教士 3 野人问题中，初始状态为（3,3,1），目标状态为（0,0,0）。

当 1 野人离开左岸到达右岸后，原状态变为 p=[(3,2,0),(3,3,1)],p[0]为当前状态，而列表最后一个为初始状态，只要当 p[0]=(0,0,0)则完成搜索。

2、节点拓展方法与合法状态判断

（1）节点拓展：通过减少和增加传教士或野人的数量来拓展节点。

当船在左岸（B=1）：①减少野人 ②减少传教士 ③减少野人和传教士

当船在右岸（B=0）：①增加野人 ②增加传教士 ③增加野人和传教士

（2）合法状态判断

①左岸传教士数量等于总数或左岸传教士为 0：C>=0,C<=N

②左岸传教士数量基于 0 到 N 之间时: C>=0,M>=C,M<=N,C<=N,N-M>=N-C

③其他状态为不合法

3、搜索过程

①建立只含有初始节点 S 的搜索图 G，把 S 放到 OPEN 表中；

②建立 CLOSED 表，其初始值为空表；

③若 OPEN 表是空表，则失败退出；

④选择 OPEN 表中第一个节点，把它从 OPEN 表移出并放进 CLOSED 表中，称此节点为节点 n；

⑤若 n 为目标节点，则有解并成功退出。

⑥沿指针追踪图 G 中从 n 到 S 这条路径得到解（指针在步骤⑦中设置）；

⑦扩展 n，生成不是 n 的祖先的那些后继节点的集合 M，把 M 的这些成员作为 n 的后继节点添入图 G 中；

对 M 中子节点进行如下处理：

-对没在 G 中出现过的（即没在 OPEN 或 CLOSED 表中出现过的）M 成员设置一个指向 n 的指针，把 M 的这些成员加进 OPEN 表；

-已在 OPEN 或 CLOSED 表中的每个 M 成员，确定是否需要更改指向 n 的指针方向；

-已在 CLOSED 表中的每个 M 成员，确定是否需要更改图 G 中它的每个后裔节点指向父节点的指针。

⑧按某种方式或按某个试探值，重排 OPEN 表；

⑨转步骤③。

1、A*算法的基本原理分析；

在图的一般搜索算法中，如果在搜索过程的步骤⑦利用估价函数 f(n)=g(n)+h(n)对 open 表中的节点进行排序，则该搜索算法为 A*算法。

g(n)：从初始节点到 n 的实际代价

因为 n 为当前节点，搜索已达到 n 点，所以 g(n)可计算出。

h(n)：启发函数，从 n 到目标节点的最佳路径的估计代价。

因为尚未找到解路径，所以 h(n)仅仅是估计值。

对 A 算法中的 g(n)和 h(n)做出限制：

g(n)>=g(n)（g*(n)为 S0 到 n 的最小费用）

-h(n)<=h*(n)（h*(n)为 n 到 Sg 的实际最小费用）

则算法被称为 A*算法。

2、传教士—野人过河问题的知识表示方法分析；

在这个问题中，需要考虑：

1、两岸的传教士人数和野人人数

2、船在左岸还是在右岸

已知：传教士和野人数：N（两者默认相同），船的最大容量：K

定义：M：左岸传教士人数 C：左岸野人人数 B：左岸船个数

可用一个三元组来表示左岸状态，即 S=(M,C,B)。
约束条件：M>=0,C>=0,B=1 或 0
已知左岸状态，右岸的状态为：
右岸传教士人数：M’=N-M
右岸野人人数：C’=N-C
右岸船数：B’=1-B
满足同样的约束条件

3、针对传教士—野人过河问题的 A*算法详细分析
（1）A*算法求解传教士和野人过河问题，主要实现过程：
①使用状态空间法将问题的求解抽象为状态空间的搜索。
②根据 A*算法的思想、A*算法的具体步骤、设计估价函数的方法，针对传教士－野人过河问题设计出估价函数 f(n)，给出条件约束函数。
（2）估价函数设计：f(n)=g(n)+M+C-K*B

h(n)=M+C-K*B，把状态转换后左岸剩余人数作为启发性信息，使 f(n)尽可能地小。
合理性分析：
本问题中，在满足条件约束的前提下，总是希望能使左岸的人数最少。当左岸有船时，应当使船每次都满负荷运载，即运 KB 人过河。
然而，在最大运载量为 K 的情况下，状态转换后左岸的剩余人数不可能小于 M+C-KB，即从节点 n 到目标节点的最小代价 h*(n)不可能小于 h(n)，因此，满足 A*算法的条件限制 h(n)<=h(n)。
（3）本问题中操作是指用船把传教士或野人从河的左岸运到右岸，或者从河的右岸运到左岸，并且每个操作都应该满足以下 3 个条件：
①船至少有一个人(M 或 C)操作，离开岸边的 M 和 C 的减少数目等于到达岸边的 M 和 C 的增加数目。
②每次操作，船上的人数不得超过 K 个。
③操作应保证不产生非法状态。

主要函数：
def GJ(this,k):#估价函数计算 h(n) = M + C - K * B
def creat(array,M,C,B,N):#判断生成节点是否符合规则、判断是否重复

主要变量：
Creatpoint //生成节点数
Searchpoint //搜索到的节点数

具体代码见附录。

5. 遗传算法解 TSP 问题

5.1 问题的分析与求解

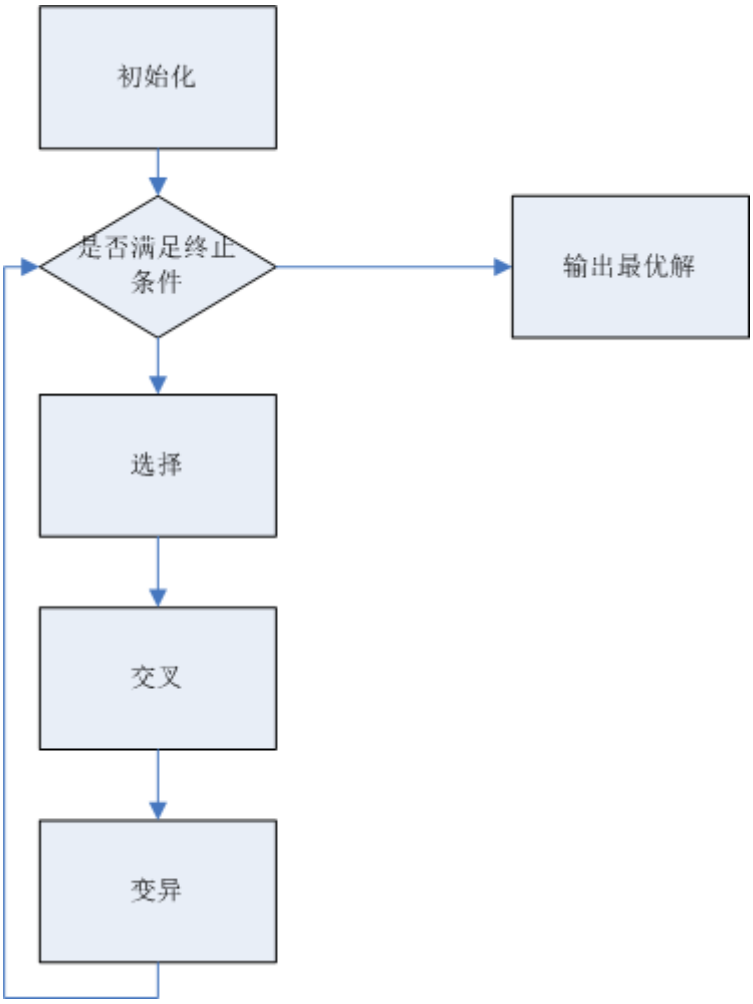
遗传算法是模仿生物遗传学和自然选择机理，通过人工方式所构造的一类优化搜索算法，是对生物进化过程进行的一种数学仿真，是进化计算的最重要的形式。

遗传算法为那些难以找到传统数学模型的难题指出了一个解决方法。

进化计算和遗传算法借鉴了生物科学中的某些知识，这也体现了人工智能这一交叉学科的特点。

用遗传算法解旅行商 TSP 问题：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，路径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。

我们使用全局变量定义城市数目，可以根据需要进行修改。
遗传算法的要点有三：1.编码与解码 2.适应度函数 3.遗传操作
其中遗传算法的遗传操作主要流程如下：



因此我们主要的函数为：

一个选择函数 void xuanze()；一个交叉（交配）函数 void jiaopei()；一个变异函数 void bianyi()。为了模拟染色体的活动，我们还得有类似染色体的结构体，结构体里放什么元素长什么样子这个我们稍后讨论。

除此之外，要解决 TSP 问题首先我们得有一个存储城市间距离的邻接矩阵，这个城市间的距离是我们随机数生成的，用二维数组存储，为完成这些事我们还需要一个初始化函数 void init()。染色体的结构要为解决 TSP 问题来确定编码规则和含义，我们希望最后能输出一条最优秀的染色体告知我们旅行商经过这 n 个城市的顺序，为选出这条最优染色体显然我们需要一个评价其好坏的函数 void pingjia()，来计算适应度值。评价函数需要根据一定的机制依据某指标评价任意一条染色体的优劣，我们把这个指标定义为染色体的存活率 p。存活率 p：总路径越小，生存概率越高，我们这里以存活率 **p=1-单条染色体的路径/种群总路径** 来衡量染色体优劣，生存概率最高的染色体将被当作最优解输出，此为我们的适应度函数的定义。

关于编码：我们理想中的这条染色体应该是如此模样：例如旅行商经过十个城市，最后的最优染色体为 1，4，6，5，0，7，2，8，9，3，意思是旅行商应当按照 1→4→6→5→0→7→2→8→9→3 走距离最短。何为最优？即让旅行商走过的路径要最短。

4.2 主要的数据结构

通过上述分析，我们基本可以确定染色体的结构体如下：

```
struct group    //染色体的结构
{
    int city[cities];//城市的顺序
    int adapt;//适应度，这里指用于计算适应度的该条染色体经过所有城市的路径总和
    double p;//在种群中的幸存概率
}group[num],grouptemp[num];
```

为什么要定义两种相同的结构体 group[num]和 grouptemp[num]呢？因为选择过程我们需要选择出能生存的个体暂存进中间变量 grouptemp[num]，然后再重新导入 group[num]，即把不能生存的个体剔除，以完成对染色体的更新。

那么现在可以定下来我们的主要变量为：

```
#define cities 10    //城市的个数
#define MAXX 100//迭代次数
#define pc 0.8 //交配概率
#define pm 0.05 //变异概率
#define num 10//种群的大小
int bestsolution;//最优染色体
int distance[cities][cities];//城市之间的距离
```

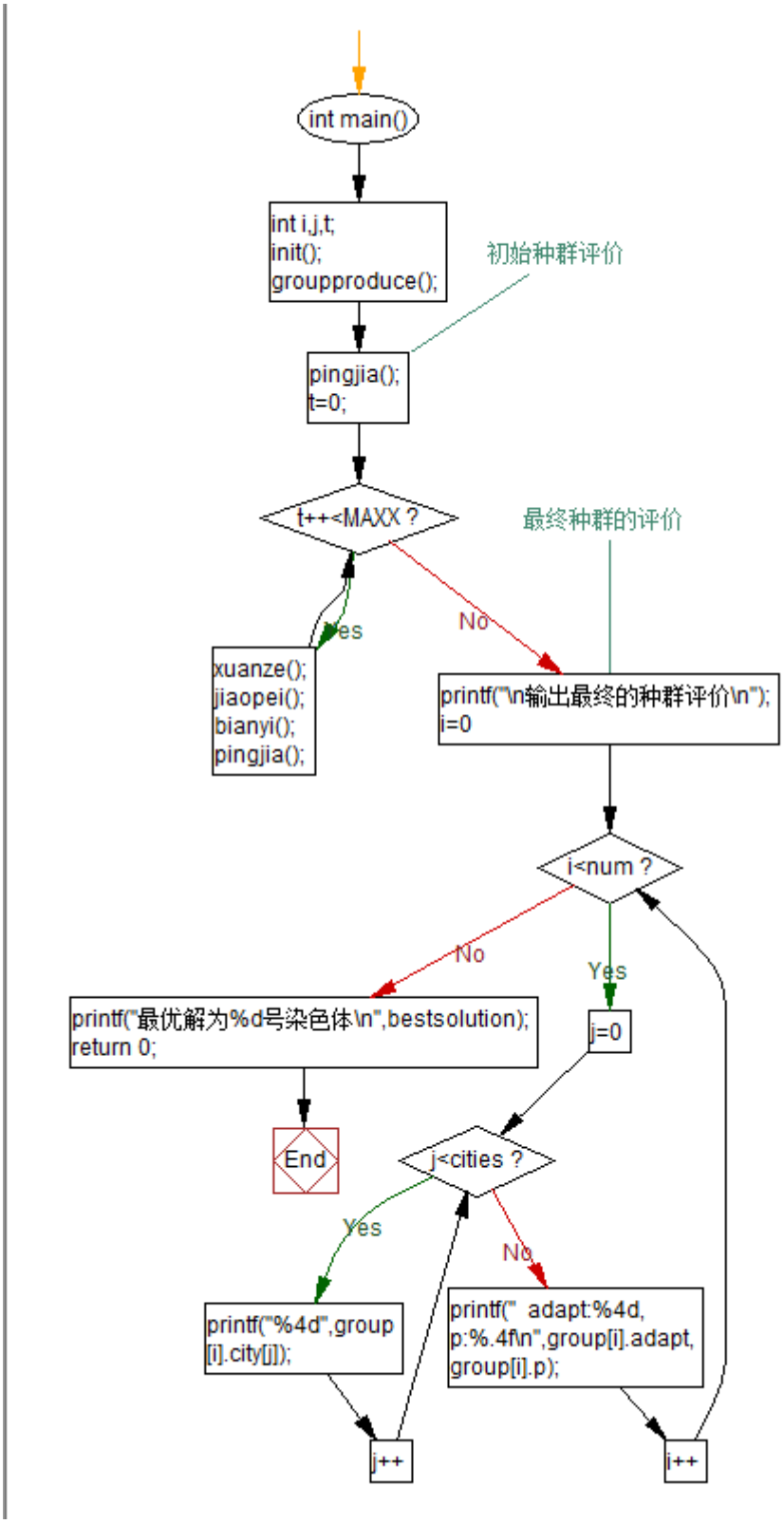
好，既然已经确定染色体的结构长什么样，要对染色体们操作就得有具体的操作对象

吧，老板，上染色体！因此我们还需要一个初始生成染色体的函数 void groupproduce()，替我们随机产生初初始染色体种群。

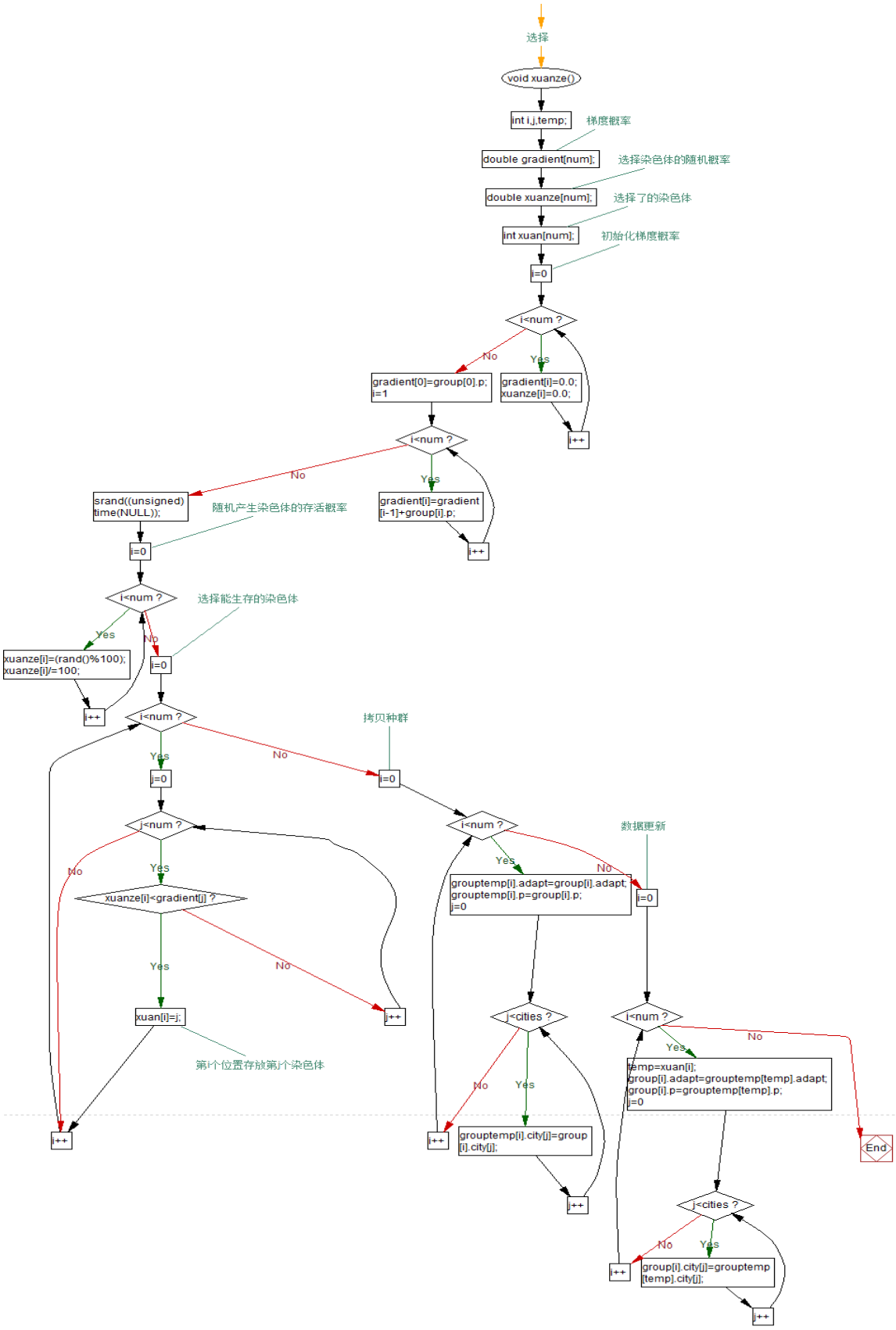
其余的更重要的事情就是染色体操作函数：一个选择函数 void xuanze()；一个交叉（交配）函数 void　jiaopei()；一个变异函数 void bianyi()要怎么写了。

4.3 主要的函数展示

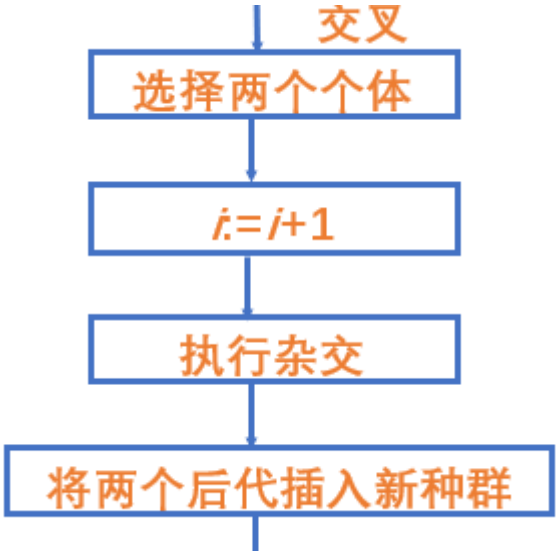
我们设想的 main()大概流程长这样了：



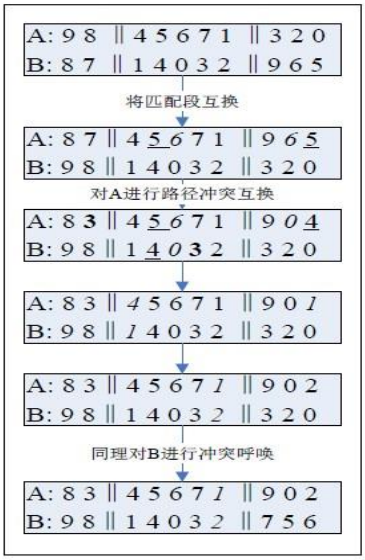
选择函数：选择函数肩负了更新种群的职责，需要把生存率不合要求的染色体剔除


选择
void xuanze()
int i,j,temp;
梯度概率
double gradient[num];
选择染色体的随机概率
double xuanze[num];
选择了的染色体
int xuan[num];
初始化梯度概率
i=0
i<num ?
No
Yes
gradient[0]=group[0].p;
i=1
gradient[i]=0.0;
xuanze[i]=0.0;
i++
i<num ?
No
Yes
gradient[i]=gradient
[i-1]+group[i].p;
i++
srand((unsigned)
time(NULL));
随机产生染色体的存活概率
i=0
i<num ?
选择能生存的染色体
Yes
No
xuanze[i]=(rand()%100);
xuanze[i]/=100;
i++
i=0
i<num ?
拷贝种群
Yes
No
j=0
i=0
j<num ?
i<num ?
数据更新
No
Yes
Yes
No
xuanze[i]<gradient[j] ?
grouptemp[i].adapt=group[i].adapt;
grouptemp[i].p=group[i].p;
j=0
i=0
Yes
No
xuan[i]=j;
j++
第i个位置存放第j个染色体
j<cities ?
i<num ?
No
Yes
No
Yes
temp=xuan[i];
group[i].adapt=grouptemp[temp].adapt;
group[i].p=grouptemp[temp].p;
j=0
End
i++
grouptemp[i].city[j]=group
[i].city[j];
j++
j<cities ?
No
Yes
i++
group[i].city[j]=grouptemp
[temp].city[j];
j++

交配函数：简单的说交配函数原理如下：



复杂了说，我们使用的是这样的交配原理：
注意，交叉后要确保每个城市有且仅走了一次



本实例中交叉采用部分匹配交叉策略，其基本实现的步骤是：

步骤1： 随机选取两个交叉点

步骤2： 将两交叉点中间的基因段互换

步骤3：将互换的基因段以外的部分中与互换后基因段中元素冲突的用另一父代的相应位置代替，直到没有冲突。

本例中路径实例如图交叉点为2，7，交换匹配段后A中冲突的有7 、6、5，在B的匹配段中找出与A匹配段中对应位置的值7-3，6-0，5-4，继续检测冲突直到没有冲突。对B 做同样操作，得到最后结果。

（图 4）

具体代码如下：

```
void    jiaopei()
{
      int i,j,k,kk;
      int t;//参与交配的染色体的个数
      int point1,point2,temp;//交配断点
      int pointnum;
      int temp1,temp2;
      int map1[cities],map2[cities];
      double jiaopeip[num];//染色体的交配概率
      int jiaopeiflag[num];//染色体的可交配情况
```

```
int kkk,flag=0;
//初始化
for(i=0;i<num;i++)
{
    jiaopeiflag[i]=0;
}
//随机产生交配概率
srand((unsigned)time(NULL));
for(i=0;i<num;i++)
{
    jiaopeip[i]=(rand()%100);
    jiaopeip[i]/=100;
}
//确定可以交配的染色体
t=0;
for(i=0;i<num;i++)
{
    if(jiaopeip[i]<pc)
    {
        jiaopeiflag[i]=1;
        t++;
    }
}
t=t/2*2;//t 必须为偶数
//产生 t/2 个 0-9 交配断点
 srand((unsigned)time(NULL));
temp1=0;
//temp1 号染色体和 temp2 染色体交配
for(i=0;i<t/2;i++)
{
    point1=rand()%cities;//交配点 1
    point2=rand()%cities;//交配点 2
    //选出一个需要交配的染色体 1
    for(j=temp1;j<num;j++)
    {
        if(jiaopeiflag[j]==1)
        {
            temp1=j;
            break;
        }
    }
    //选出另一个需要交配的染色体 2 与 1 交配
    for(j=temp1+1;j<num;j++)
    {
```

```
        if(jiaopeiflag[j]==1)
        {
            temp2=j;
            break;
        }
    }
    //进行基因交配
    if(point1>point2) //保证 point1<=point2
    {
        temp=point1;
        point1=point2;
        point2=temp;
    }
    //初始化
     memset(map1,-1,sizeof(map1));
    memset(map2,-1,sizeof(map2));
    //断点之间的基因产生映射
    for(k=point1;k<=point2;k++)
    {
        map1[group[temp1].city[k]]=group[temp2].city[k];
        map2[group[temp2].city[k]]=group[temp1].city[k];
    }
    //断点两边的基因互换
    for(k=0;k<point1;k++)
    {
        temp=group[temp1].city[k];
        group[temp1].city[k]=group[temp2].city[k];
        group[temp2].city[k]=temp;
    }
    for(k=point2+1;k<cities;k++)
    {
        temp=group[temp1].city[k];
        group[temp1].city[k]=group[temp2].city[k];
        group[temp2].city[k]=temp;
    }
    printf("处理冲突----------\n");
    //处理染色体 1 产生的冲突基因
    for(k=0;k<point1;k++)
    {
        for(kk=point1;kk<=point2;kk++)
        {
            if(group[temp1].city[k]==group[temp1].city[kk])
            {
                group[temp1].city[k]=map1[group[temp1].city[k]];
```

```
                //find
                for(kkk=point1;kkk<=point2;kkk++)
                {
                    if(group[temp1].city[k]==group[temp1].city[kkk])
                    {
                        flag=1;
                        break;
                    }
                }
                if(flag==1)
                {
                    kk=point1-1;
                    flag=0;
                }
                else
                {
                    flag=0;
                    break;
                }
            }
        }

}
for(k=point2+1;k<cities;k++)
{
    for(kk=point1;kk<=point2;kk++)
    {
        if(group[temp1].city[k]==group[temp1].city[kk])
        {
            group[temp1].city[k]=map1[group[temp1].city[k]];
            //find
            for(kkk=point1;kkk<=point2;kkk++)
            {
                if(group[temp1].city[k]==group[temp1].city[kkk])
                {
                    flag=1;
                    break;
                }
            }
            if(flag==1)
            {
                kk=point1-1;
                flag=0;
            }
```

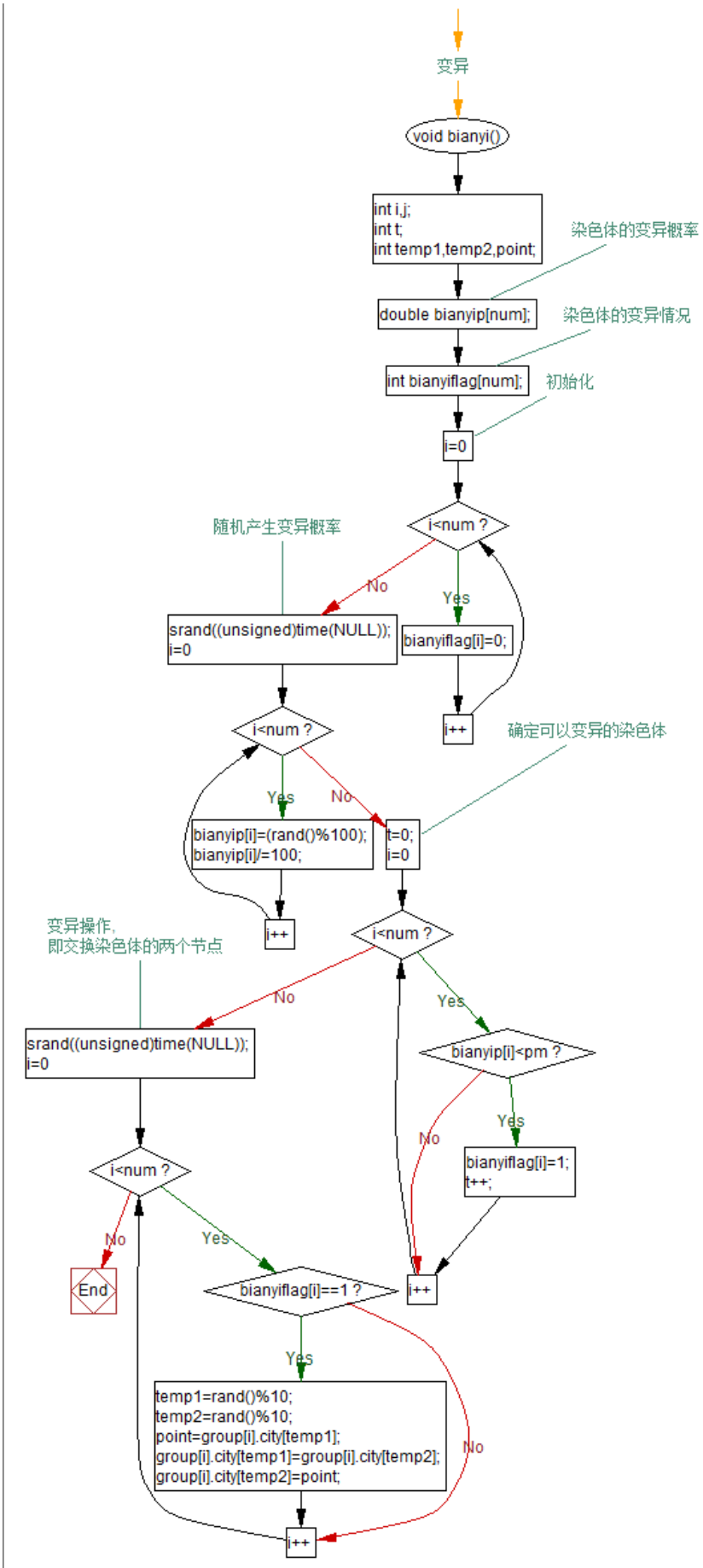
```
                else
                {
                    flag=0;
                    break;
                }
            }
        }
    }
    //处理 2 染色体产生的冲突基因
    for(k=0;k<point1;k++)
    {
        for(kk=point1;kk<=point2;kk++)
        {
            if(group[temp2].city[k]==group[temp2].city[kk])
            {
                group[temp2].city[k]=map2[group[temp2].city[k]];
                //find
                for(kkk=point1;kkk<=point2;kkk++)
                {
                    if(group[temp2].city[k]==group[temp2].city[kkk])
                    {
                        flag=1;
                        break;
                    }
                }
                if(flag==1)
                {
                    kk=point1-1;
                    flag=0;
                }
                else
                {
                    flag=0;
                    break;
                }
            }
        }
    }
    for(k=point2+1;k<cities;k++)
    {
        for(kk=point1;kk<=point2;kk++)
        {
            if(group[temp2].city[k]==group[temp2].city[kk])
            {
```

```
                    group[temp2].city[k]=map2[group[temp2].city[k]];
                    //find
                    for(kkk=point1;kkk<=point2;kkk++)
                    {
                        if(group[temp2].city[k]==group[temp2].city[kkk])
                        {
                            flag=1;
                            break;
                        }
                    }
                    if(flag==1)
                    {
                        kk=point1-1;
                        flag=0;
                    }
                    else
                    {
                        flag=0;
                        break;
                    }
                }
            }
        }
         temp1=temp2+1;
    }
}
```

变异函数：

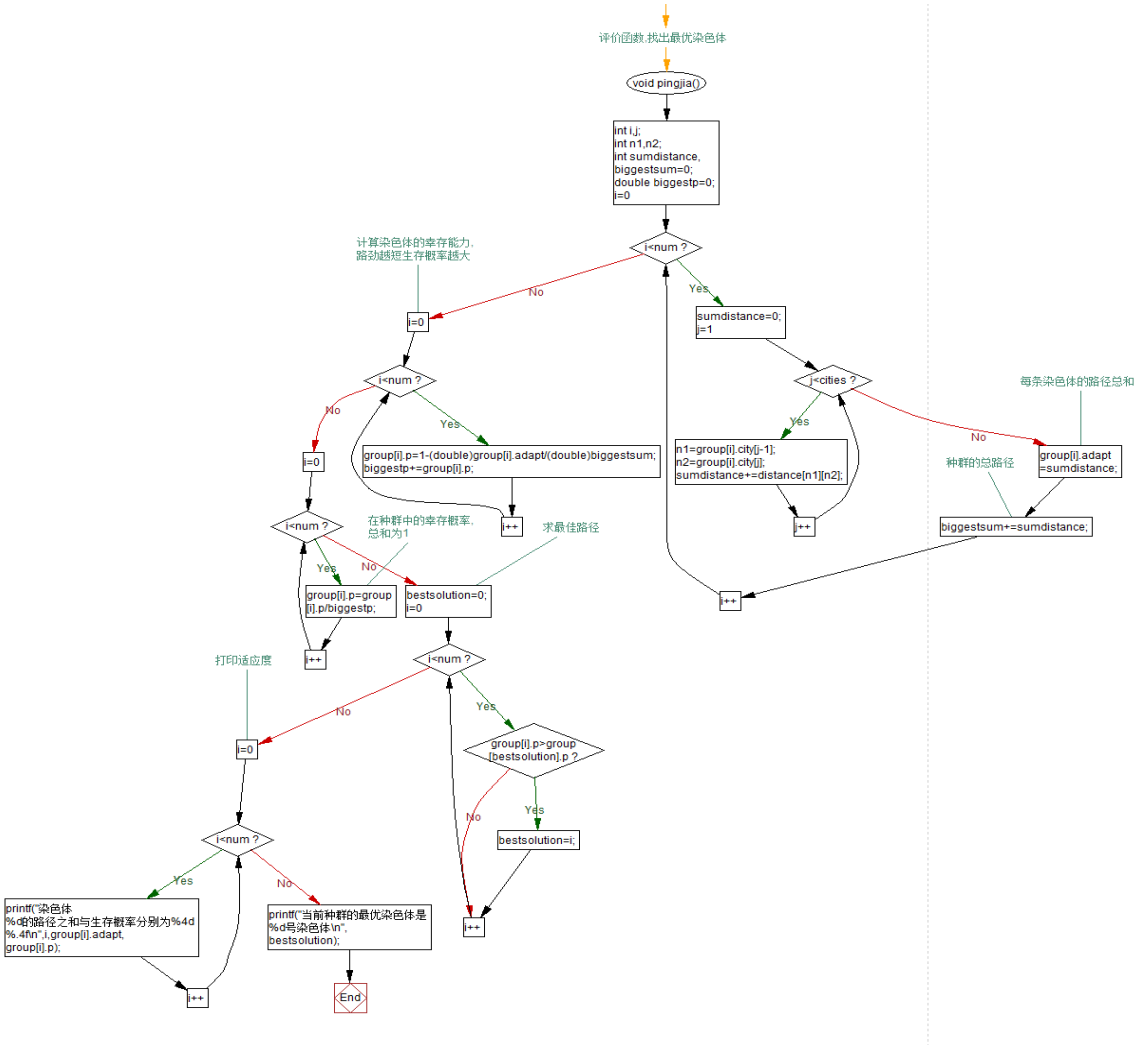
这是随机选一个变异概率达到要求的染色体，随机选个染色体上的两个节点（城市）进行基因交换的函数，也就是换了下城市的经过顺序。为平平无奇的寻找最优路径之旅增加点波澜。

流程图如下：


变异
void bianyi()
int i,j;
int t;
int temp1,temp2,point;
染色体的变异概率
double bianyip[num];
染色体的变异情况
int bianyiflag[num];
初始化
i=0
i<num ?
随机产生变异概率
No
Yes
srand((unsigned)time(NULL));
i=0
bianyiflag[i]=0;
i++
i<num ?
确定可以变异的染色体
Yes
No
bianyip[i]=(rand()%100);
bianyip[i]/=100;
t=0;
i=0
i++
i<num ?
变异操作，
即交换染色体的两个节点
No
Yes
srand((unsigned)time(NULL));
i=0
bianyip[i]<pm ?
Yes
No
bianyiflag[i]=1;
t++;
i<num ?
No
Yes
End
bianyiflag[i]==1 ?
i++
Yes
temp1=rand()%10;
temp2=rand()%10;
point=group[i].city[temp1];
group[i].city[temp1]=group[i].city[temp2];
group[i].city[temp2]=point;
No
i++

评价函数：

评价函数，起到了计算适应度函数，输出最终结果的作用。我们开始初始化随机生成染色体时评价一次，每轮迭代评价一次。最后一轮输出最终评价时不再使用该函数。假设迭代次数为 100，染色体种群大小为 10，那么第 100 次迭代后的 10 条染色体为运行该程序得到的前 10 条最优解，在这前 10 名中寻找出第 1 名的染色体即为我们要求的最优解了。



四、上机实验结果截图（关键界面）

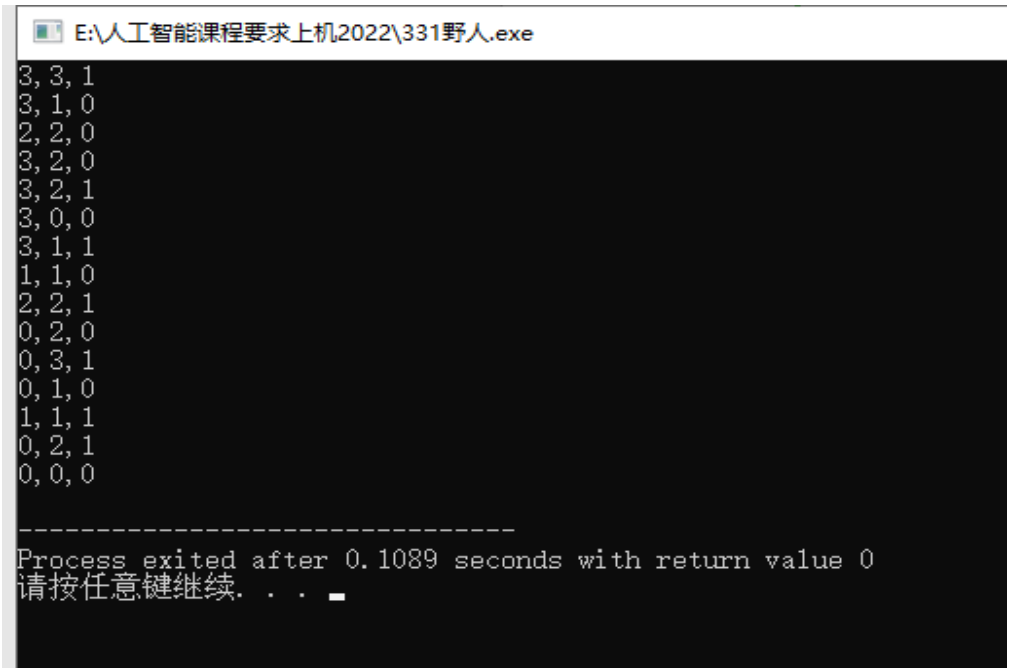


图 1　331 野人与传教士求最优解运行结果

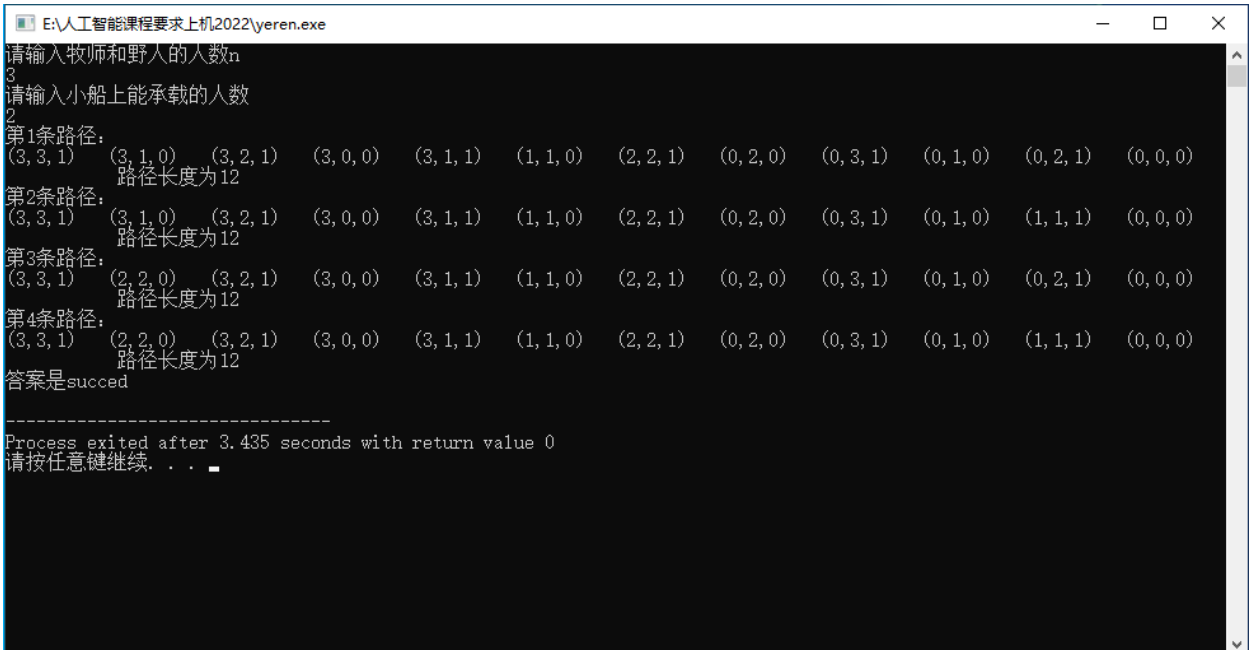
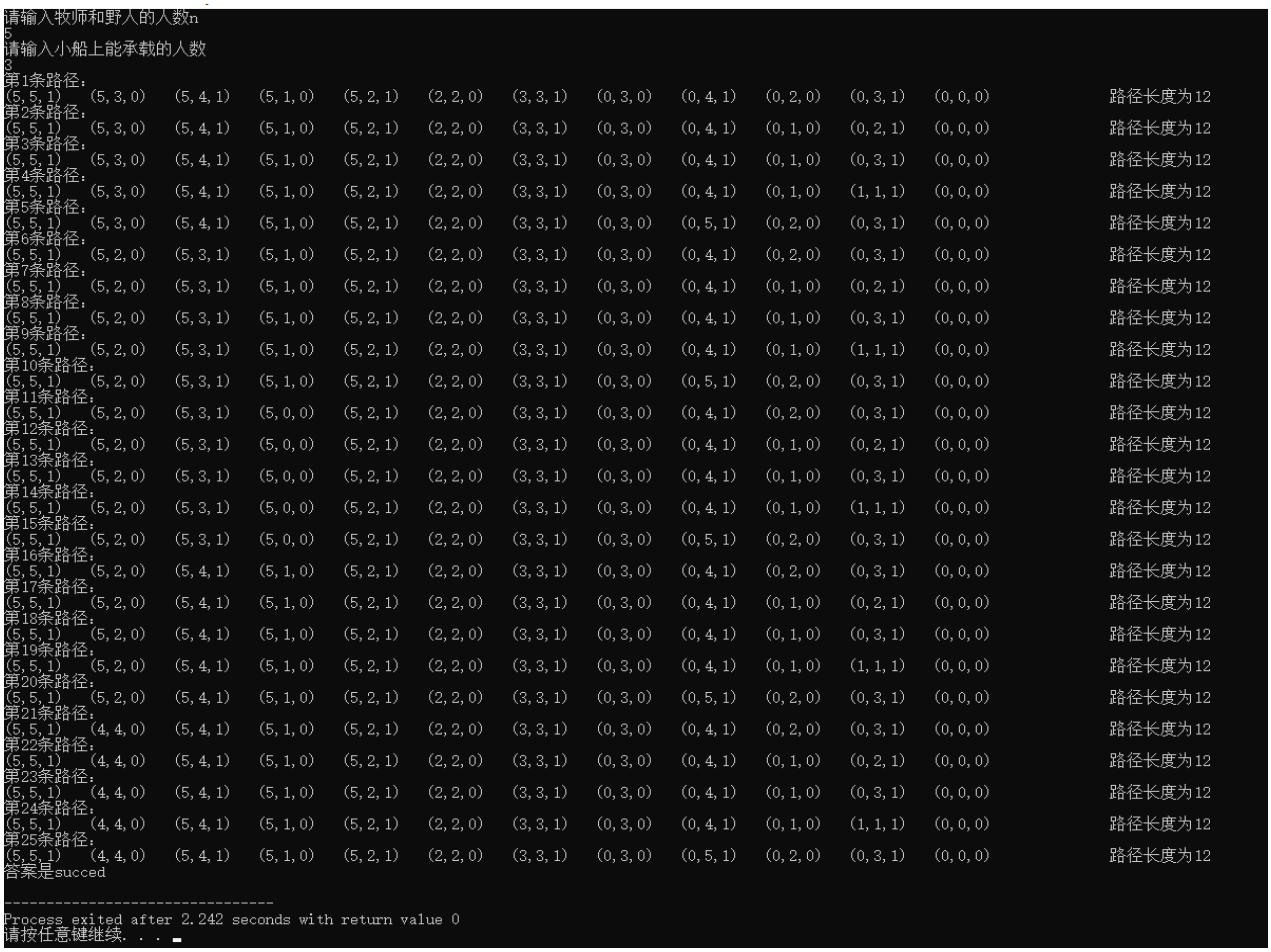
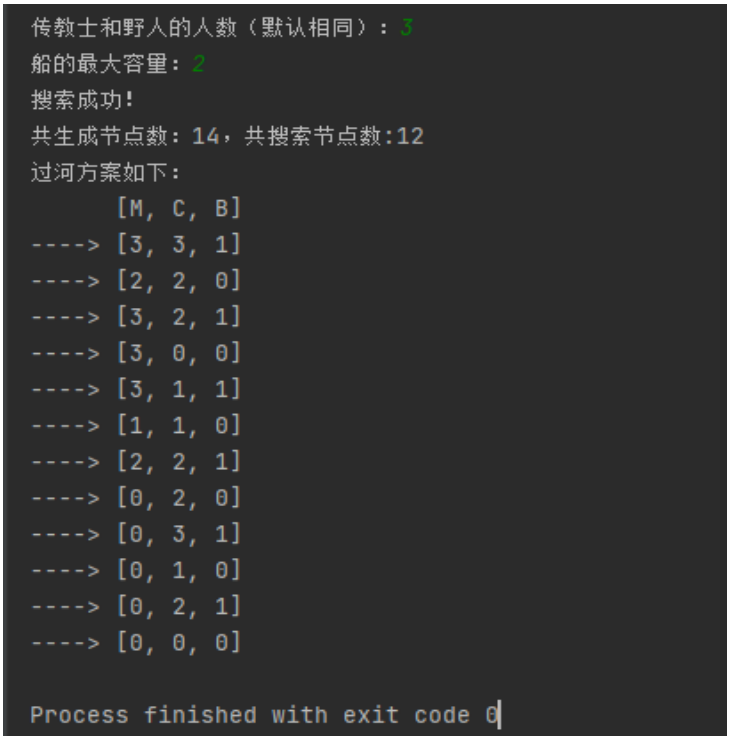


图 2　　NNk 野人与传教士渡河 3-3-2 运输规则的所有路径



图表 3　N-N-K 野人与传教士渡河 5-5-3 运输规则的所有路径



图表 4 N-N-K 野人与传教士渡河 3-3-2 运输规则的最优路径

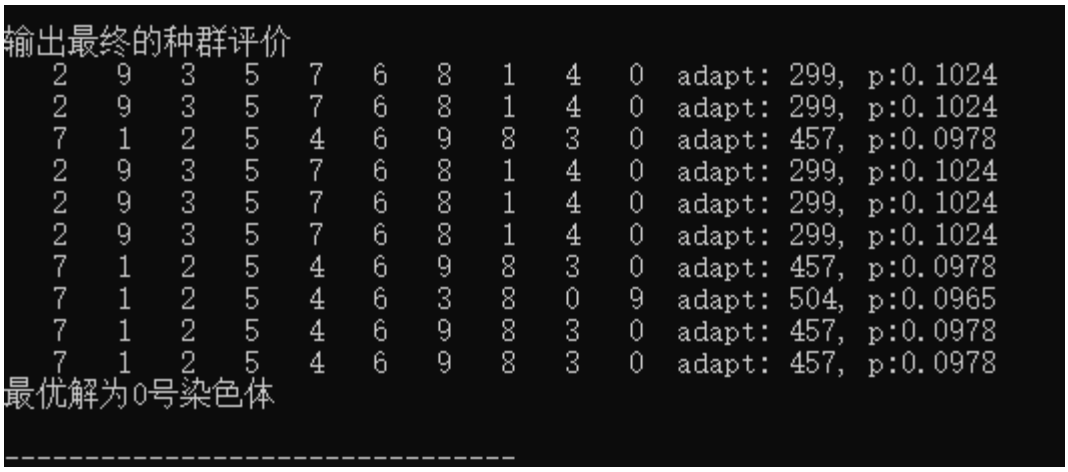
```
传教士和野人的人数（默认相同）：5
船的最大容量：3
搜索成功!
共生成节点数：25，共搜索节点数:17
过河方案如下：
      [M, C, B]
----> [5, 5, 1]
----> [4, 4, 0]
----> [5, 4, 1]
----> [5, 1, 0]
----> [5, 2, 1]
----> [2, 2, 0]
----> [3, 3, 1]
----> [0, 3, 0]
----> [0, 4, 1]
----> [0, 1, 0]
----> [0, 2, 1]
----> [0, 0, 0]

Process finished with exit code 0
```

图表 5　N-N-K 野人与传教士渡河 5-5-3 运输规则的最优路径

```
城市的距离矩阵如下
   0  12  46  33  58  65  98  35  79  76
  12   0  59  79   0  85  78  27  14  90
  46  59   0  31  45  17  70  32  88  18
  33  79  31   0  54  55  66  98  79  19
  58   0  45  54   0  41  60  67  32  15
  65  85  17  55  41   0   8  46  24  67
  98  78  70  66  60   8   0  85   4  49
  35  27  32  98  67  46  85   0  96  26
  79  14  88  79  32  24   4  96   0  92
  76  90  18  19  15  67  49  26  92   0
初始的种群
   2   6   3   8   5   9   0   7   4   1
   5   7   0   2   8   4   6   9   1   3
   4   9   1   6   2   5   7   8   3   0
   7   5   2   0   8   1   6   3   9   4
   8   7   4   3   2   1   5   6   0   9
   2   9   3   5   7   6   8   1   4   0
   7   1   0   2   8   3   4   9   5   6
   7   4   0   1   2   9   8   6   3   5
   2   3   7   5   9   8   6   0   1   4
   0   8   3   4   2   6   9   1   7   5
染色体0的路径之和与生存概率分别为 484  0.0994
染色体1的路径之和与生存概率分别为 525  0.0984
染色体2的路径之和与生存概率分别为 524  0.0985
染色体3的路径之和与生存概率分别为 380  0.1019
染色体4的路径之和与生存概率分别为 574  0.0972
染色体5的路径之和与生存概率分别为 299  0.1039
染色体6的路径之和与生存概率分别为 396  0.1015
染色体7的路径之和与生存概率分别为 431  0.1007
染色体8的路径之和与生存概率分别为 448  0.1003
染色体9的路径之和与生存概率分别为 539  0.0981
当前种群的最优染色体是5号染色体
处理冲突----------------------
处理冲突----------------------
处理冲突----------------------
处理冲突----------------------
染色体0的路径之和与生存概率分别为 542  0.0976
染色体1的路径之和与生存概率分别为 603  0.0960
染色体2的路径之和与生存概率分别为 470  0.0994
染色体3的路径之和与生存概率分别为 260  0.1046
染色体4的路径之和与生存概率分别为 434  0.1003
染色体5的路径之和与生存概率分别为 539  0.0976
染色体6的路径之和与生存概率分别为 437  0.1002
染色体7的路径之和与生存概率分别为 431  0.1003
染色体8的路径之和与生存概率分别为 431  0.1003
染色体9的路径之和与生存概率分别为 299  0.1036
当前种群的最优染色体是3号染色体
处理冲突----------------------
处理冲突----------------------
处理冲突----------------------
处理冲突----------------------
```

图表 6　遗传算法解 TSP 问题运行结果（1）



图表 7　遗传算法解 TSP 问题运行结果（2）

五、总结体会

5.1 野人与传教士渡河问题的体会

理解了深度优先搜索、广度优先搜索等盲目搜索算法的使用和启发性算法之 A*算法的使用，在接触以前以为这些算法必定是毫无关联的，在接触以后发现其实只需要在代码上稍作改动就变成了另外一种算法，比如说深度优先搜索只需要把广度优先中对 open 表的访问由先进后出改为先进先出，而 A*算法相比于宽度优先搜索也只多了一步对 open 表按照估值函数进行排序。

但是目前来看只能说是理解了在这道题里如何去使用这些算法，想在换一道题的情况下成功用出这些遍历搜索算法还要有一定的练习时间。

5.2 对遗传算法解 TSP 问题的体会

对于遗传算法，这学期数学建模两次比赛接触过，但是不会用，都是生搬硬套理论在那儿强行使用进实际问题，编码基本没有什么逻辑，这次通过 TSP 问题这个较为简单经典的算法问题终于看懂怎么用进来了，以后会融会贯通多加使用。个人感觉遗传算法的难点在如何把实际问题合理编码进染色体，以及如何确认适应度函数，至于遗传算法三个遗传操作都是套路，直接照搬即可。

但是遗传算法不是万能的，它有以下缺点：

（1）全局搜索能力不强,很容易陷入局部最优解跳不出来;

（2）将遗传算法用于解决各种实际问题后，人们发现遗传算法也会由于各种原因过早向目标函数的局部最优解收敛，从而很难找到全局最优解。

（3）遗传算法中评价个体生存适应度函数如果设计不当也会存在一些问题：适应度函数可能遇到初期部分较突出的个体主导选择过程，在进化末期由于个体差异太小导致陷入局部极值的问题。

遗传算法适合小规模 TSP 问题的分析求解，其迭代次数与个体数目成正比，个

体数目越多，则需要更多次迭代才能得到最优解。

5.3 对于作业中神经网络的体会

作业相比于上机问题而言我认为更有吸引力，也让我付出了更多时间泡在图书馆里去了解什么是多层前馈型神经网络，什么是向前传播和向后传播。上机题属于贴近我们水平的实践，而作业则是拓展了我们的眼界。有些算法课上听不懂，课下去看或者多换几本书来看就一目了然了，比如蚁群算法和模拟退火算法等，但是大多都是理论，要想应用到实例中还要多看实例和自己练习。

按理说在数学建模里神经网络模型用到的时候比遗传算法好像要多一些，但是却不太了解，大多时候靠 matlab 工具箱解决完就认为完事了，直到亲自去理解了那些传播过程，看运用实例和数学理论讲解才终于明白了其中的数学原理和算法原理。神经网络和遗传算法我认为是人工智能这本书的精髓，它让我看到复杂算法的确可以模拟生物的行为模式来解决实际问题，不由得感慨发明这些算法的数学天才脑洞太大，也明白人工智能进展缓慢确实是问题难度到了一定程度，想要拿来解决实际问题的话，发展不慢都难。

六、参考文献

[1] 蔡自兴.人工智能及其应用[M].北京：清华大学出版社，2016.07.

[2] 卢启衡,冯晓红.基于宽度优先搜索的路径生成算法[J].现代计算机,2006(12):87-89.

[3]mylovestart.遗传算法解决 TSP 问

题.https://blog.csdn.net/mylovestart/article/details/8977005#cpp

[4]Davy_Zhuang. A*算法解决传教士—野人过河问题.

https://blog.csdn.net/u013300280/article/details/106234521?spm=1001.2014.3001.5506

[5]pansong291PS.从九宫问题浅谈广度优先搜索与深度优先搜索策

略.https://blog.csdn.net/pansong291PS/article/details/83060552?spm=1001.2014.3001.5506

七、附录

7.1 野人与传教士宽度优先算法求最优解代码 C++：

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

typedef struct
```

```
{
    int m;//表示传教士
    int c;// 表示野人
    int b;//船状态,1 为此岸，0 为不在起始子岸
}MCNode;

list<MCNode> open;//相当于队列
vector<MCNode> closed;//closed 表

//判断是否是目标结点
bool IsGoal(MCNode tNode)
{
    if(tNode.m==0&&tNode.c==0&&tNode.b==0)
        return true;
    else
        return false;
}
//判断是否是合法状态
bool IsLegal(MCNode tNode)
{
    if(tNode.m>=0&&tNode.m<=3&&tNode.c>=0&&tNode.c<=3)
    {
        if((tNode.m==tNode.c)||(tNode.m==3)||(tNode.m==0))//事实上如果出现【2，1，1】这种情况对岸已经是不合法的了，合法只有这三种情况
            return true;
        else
            return false;
    }
    else
        return false;
}
//重载运算符，判断两结构体是否相等
bool operator==(MCNode m1,MCNode m2)
{
    if(m1.m==m2.m&&m1.c==m2.c&&m1.b==m2.b)
        return true;
    else
        return false;
}
//判断是否已在 closed 表中
bool IsClosed(MCNode tNode)
{
    int i;
    for(i=0;i!=closed.size();i++)
    {
```

```
51.         if(tNode==closed[i])
52.             return true;
53.     }
54.     if(i==closed.size())
55.         return false;
56. }
57. void ExpandNode(MCNode tNode,int b,list<MCNode> &open)
58. {
59.     MCNode node[5];//应用 5 条规则集生成新结点
60.     if(b==1)
61.     {
62.         for(int i=0;i<5;i++)
63.             node[i].b=0;
64.         node[0].m=tNode.m-2;
65.         node[0].c=tNode.c;
66.         node[1].m=tNode.m;
67.         node[1].c=tNode.c-2;
68.         node[2].m=tNode.m-1;
69.         node[2].c=tNode.c-1;
70.         node[3].m=tNode.m-1;
71.         node[3].c=tNode.c;
72.         node[4].m=tNode.m;
73.         node[4].c=tNode.c-1;
74.     }
75.     else
76.     {
77.         for(int i=0;i<5;i++)
78.             node[i].b=1;
79.         node[0].m=tNode.m+2;
80.         node[0].c=tNode.c;
81.         node[1].m=tNode.m;
82.         node[1].c=tNode.c+2;
83.         node[2].m=tNode.m+1;
84.         node[2].c=tNode.c+1;
85.         node[3].m=tNode.m+1;
86.         node[3].c=tNode.c;
87.         node[4].m=tNode.m;
88.         node[4].c=tNode.c+1;
89.     }
90.     for(int i=0;i<5;i++)
91.         if(IsLegal(node[i])&&!IsClosed(node[i]))
92.             //open.push_front(node[i]);//队列后进先出，深度优先搜索，最后得到一条最小解序列
93.             open.push_back(node[i]);//队列后进后出，宽度优先搜索，最后得到最小解序列状态空间图
```

```
}
 int main()
{
    MCNode InitNode,unode;
    InitNode.m=3;
    InitNode.c=3;
        InitNode.b=1;
        open.push_back(InitNode);//将初始状态空间加入到队列
        while(!open.empty())
        {
            unode=open.front();
            open.pop_front();
            if(IsGoal(unode))
            {
                closed.push_back(unode);
                for(int i=0;i!=closed.size();i++)
                    cout<<closed[i].m<<","<<closed[i].c<<","<<close
d[i].b<<endl;
                break;
            }
            if(!IsClosed(unode))
            {
                closed.push_back(unode);
                ExpandNode(unode,unode.b,open);
            }
        }
        return 0;
    }
```

7.2 野人与传教士渡河 NNK 问题求所有路径宽度优先搜索代码 C++：

```
//基于广度优先搜索求解牧师和野人的问题
#include <iostream>
#include<queue>
//#include<Query.h>
using namespace std;

struct States
{
    int X1;//起始岸上的牧师人数
    int X2;//起始岸上的野人人数
    int X3;//小船现在位置(1 表示起始岸，0 表示目的岸)
    int tree_depth = 0;//该状态第一次出现的最低层
```

```
    States * pre;//指向前驱的指针
};
States * hasgone[1000];//已经走过的状态,用于避免以后重复
int num = 0;//hasgone 的数量
int n = 0;//牧师和野人的总人数
int c = 0;//小船上可承载人数
queue<States *> que;//队列，用于进行广度优先搜索
//保存路径，以及它对应的船来回次数
States *answer[1000];
int num_answer = 0;
int step_num[1000];//路径对应的步数

//int tree_depth = 0;
queue<States*> rubbish;//垃圾队列,用来存放最后运算完未 delete 的数据


States* Start2Destination(States* pre,int x1, int x2)//now 当前状态,从开始到目的地(x1 是船上牧师人数，x2 是船上野人人数)
{
    int Start_Priests= pre->X1 - x1;//现在岸上的牧师人数
    int Start_savages= pre->X2 - x2;//现在岸上的野人人数
    int End_Priests = n-pre->X1 + x1;//现在目的地的牧师人数
    int End_savages = n-pre->X2 +x2;//现在目的地的野人人数
    if ((x1 >= x2||x1==0)&& (Start_Priests >= Start_savages|| Start_Priests==0) &&( End_Priests >= End_savages|| End_Priests==0))//符合要求
    {
        States * now = new States;
        now->pre = pre;
        now->X1 = Start_Priests;
        now->X2 = Start_savages;
        now->tree_depth = pre->tree_depth + 1;
        now->X3 = 0;
        return now;
    }
    return NULL;
}

States* Destination2Start(States* pre,int x1, int x2)//从目的地到开始地(x1 是船上牧师人数，x2 是船上野人人数)
{
    int Start_Priests = pre->X1 +x1;//现在岸上的牧师人数
    int Start_savages = pre->X2 + x2;//现在岸上的野人人数
    int End_Priests = n - pre->X1 - x1;//现在目的地的牧师人数
    int End_savages = n - pre->X2 - x2;//现在目的地的野人人数
```

```
    if ((x1 >= x2 || x1 == 0) && (Start_Priests >= Start_savages || Start_Priests == 0) && (End_Priests >= End_savages || End_Priests == 0))//符合要求
    {
        States * now = new States;
        now->pre = pre;
        now->X1 = Start_Priests;
        now->X2 = Start_savages;
        now->tree_depth = pre->tree_depth + 1;
        now->X3 = 1;
        return now;

    }
    return NULL;
}

void showSolution(States * states, int time)
{
    if (states == NULL)
    {
        step_num[num_answer] = time;
        cout << endl;
    }
    else
    {
        showSolution(states->pre, ++time);
        cout << "(" << states->X1 << "," << states->X2 << "," << states->X3 << ")   ";
    }
}

bool Hasgone(States* one)//判断这个是否前面已经走过的
{
    for (int i = 0; i < num; i++)
    {
        if (one->X1 == hasgone[i]->X1&&one->X2 == hasgone[i]->X2&&one->X3 == hasgone[i]->X3&&one->tree_depth>hasgone[i]->tree_depth)//有重复的,且不再同一层,删除 new 的东西,return,true
        {
            delete one;

            return true;
        }
    }
    //该状态之前未走过
```

```
    //加入 hasgone 里面并 num++
    hasgone[num] = one;
    num++;
    return false;
}
int route = 0;
    void Nextstep()
    {
        if (que.empty()) return;//队列为空
        States*pre = que.front();
        rubbish.push(pre);
        que.pop();
        if (pre->X1 == 0 && pre->X2 == 0 && pre->X3 == 0)//路程已经
走完了
        {
            route++;
            cout << "第"<<route<<"条路径: ";
            showSolution(pre,0);
            cout << "            路径长度为
" << step_num[num_answer] << endl;
            answer[num_answer] = pre;
            num_answer++;
            Nextstep();
        }
        //if (Hasgone(pre)) return;//这个状态之前走过了,返回
        if (pre->X3 == 1)//在起始岸
        {
            int Max1 = pre->X1 > c ? c : pre->X1;
            int Max2 = pre->X2 > c ? c : pre->X2;
            for (int i = 0; i <= Max1; i++)
            {
                for (int j = 0; j <= Max2; j++)
                {
                    if (i + j <= c&&i+j>=1)
                    {
                        States* now=Start2Destination(pre,i,j);
                        if (now!=NULL&&!Hasgone(now))
                        {
                            que.push(now);
                        }
                    }
                }
            }
            Nextstep();
        }
```

```
137.        else//在目的岸
138.        {
139.            int Max1 = n - pre->X1 > c ? c : n-pre->X1;
140.            int Max2 = n - pre->X2 > c ? c : n-pre->X2;
141.            for (int i = 0; i <= Max1; i++)
142.            {
143.                for (int j = 0; j <= Max2; j++)
144.                {
145.                    if (i + j <= c && i + j >= 1)
146.                    {
147.                        States* now = Destination2Start(pre, i, j);
148.                        if (now != NULL && !Hasgone(now))
149.                        {
150.                            que.push(now);
151.                        }
152.                    }
153.                }
154.            }
155.            Nextstep();
156.        }
157.    }
158.
159.
160.
161.    void delete_rubbish()
162.    {
163.        while (!rubbish.empty())
164.        {
165.            States*del = rubbish.front();
166.            rubbish.pop();
167.            delete del;
168.        }
169.    }
170.
171.    int main()
172.    {
173.        cout << "请输入牧师和野人的人数 n"<<endl;
174.        cin >> n;
175.        cout << "请输入小船上能承载的人数" << endl;
176.        cin >> c;
177.        States* initial = new States();
178.        initial->X1 = n;
179.        initial->X2 = n;
180.        initial->X3 = 1;
```

```
        initial->tree_depth = 0;
        initial->pre = NULL;
        que.push(initial);
        Hasgone(initial);
        Nextstep();
        if (num_answer == 0) cout << "答案是 false" << endl;
        else cout << "答案是 succed" << endl;
        delete_rubbish();
```

7.3 野人与传教士渡河 NNk 问题 A*算法求最优路径宽度优先搜索 Python 代码：

```
def GJ(this,k):#估价函数计算 h(n) = M + C - K * B
    return this[0] + this[1] - k * this[2]

def creat(array,M,C,B,N):#判断生成节点是否符合规则、判断是否重复
    P = array[:]
    if M == N :#左岸传教士数量等于总数
        if  C >=0  and  C <= N :
            P.insert(0,[M,C,1-B])
            for i in open:
                if P[0] == i[0]:
                    return False
            for i in closed:
                if P[0] == i[0]:
                    return False
            open.append(P)
            return True
        else:
            return False
    elif M > 0 :#左岸传教士数量基于 0 到 N 之间时
        if  C >=0 and M >= C and M <= N and C <= N and N-M >= N-C:

            P.insert(0,[M,C,1-B])
            for i in open:
                if P[0] == i[0]:
                    return False
            for i in closed:
                if P[0] == i[0]:
                    return False
            open.append(P)
            return True
        else:
            return False
    elif M == 0:#左岸传教士为 0
```

```
        if  C >= 0 and C <= N:
            P.insert(0, [M, C, 1 - B])
            for i in open:
                if P[0] == i[0]:
                    return False
            for i in closed:
                if P[0] == i[0]:
                    return False
            open.append(P)
            return True
        else:
            return False
    else:
        return False

if  __name__ == '__main__':
    N = int(input("传教士和野人的人数（默认相同）："))
    K =int(input("船的最大容量："))
    open = []  #创建 open 表
    closed = [] #创建 closed 表
    sample = [N,N,1] #初始状态
    goal = [0,0,0]#目标状态
    open.append([sample])
    creatpoint = searchpoint = 0
    while(1):
        if sample == goal:
            print("初始状态为目标状态！")
            break
        if len(open) == 0:
            print("未搜索到解！")
            break
        else:
            this = open.pop(0)
            closed.append(this)
            if this[0] == goal:
                print("搜索成功！")
                print('共生成节点数：{}，共搜索节点
数:{}'.format(creatpoint,searchpoint + 1))
                print('过河方案如下：')
                print('      [M, C, B]')
                for i in this[::-1]:
                    print('---->',i)
                exit()
            #扩展节点
            searchpoint += 1
```

```
            if this[0][2] == 1 :#船在左岸时
                for i in range(1,K+1):#只
                    if creat(this,this[0][0]-i,this[0][1],this[0][2],N):
                        creatpoint += 1
                for i in range(1,K+1):
                    if creat(this,this[0][0],this[0][1]-i,this[0][2],N):
                        creatpoint += 1
                for i in range(1,K):
                    for r in range(1,K-i+1):
                        if creat(this,this[0][0] - i,this[0][1] - r, this[0][2],N):
                            creatpoint += 1
            else:#船在右岸时
                for i in range(1,K+1):
                    if creat(this,this[0][0]+i,this[0][1],this[0][2],N):
                        creatpoint += 1
                for i in range(1,K+1):
                    if creat(this,this[0][0],this[0][1]+ i,this[0][2],N):
                        creatpoint += 1
                for i in range(1,K):
                    for r in range(1,K-i+1):
                        if creat(this,this[0][0] + i,this[0][1] + r, this[0][2],N):
                            creatpoint += 1

                #计算估计函数 h(n) = M + C - K * B 重排 open 表
                for x in range(0,len(open)-1):
                    m = x
                    for y in range(x+1,len(open)):
                        if  GJ(open[x][0],K) >  GJ(open[y][0],K):
                            m = y
                    if m != x:
                        open[x],open[m] = open[m],open[x]
```

7.4 遗传算法解 TSP 问题 C++代码：

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
```

```
#include<math.h>
#include<time.h>
#define cities 10  //城市的个数
#define MAXX 100//迭代次数
#define pc 0.8 //交配概率
#define pm 0.05 //变异概率
#define num 10//种群的大小
int bestsolution;//最优染色体
int distance[cities][cities];//城市之间的距离
struct group  //染色体的结构
{
    int city[cities];//城市的顺序
    int adapt;//染色体路径总和
    double p;//在种群中的幸存概率
}group[num],grouptemp[num];
//随机产生 cities 个城市之间的相互距离
void init()
{
    int i,j;
    memset(distance,0,sizeof(distance));
    srand((unsigned)time(NULL));
    for(i=0;i<cities;i++)
    {
        for(j=i+1;j<cities;j++)
        {
           distance[i][j]=rand()%100;
           distance[j][i]=distance[i][j];
        }
    }
    //打印距离矩阵
    printf("城市的距离矩阵如下\n");
    for(i=0;i<cities;i++)
    {
        for(j=0;j<cities;j++)
        printf("%4d",distance[i][j]);
        printf("\n");
    }
}
//随机产生初试群
void groupproduce()
{
    int i,j,t,k,flag;
    for(i=0;i<num;i++)  //初始化
    for(j=0;j<cities;j++)
    group[i].city[j]=-1;
```

```
    srand((unsigned)time(NULL));
    for(i=0;i<num;i++)
    {
        //产生 10 个不相同的数字
        for(j=0;j<cities;)
        {
            t=rand()%cities;
            flag=1;
            for(k=0;k<j;k++)
            {
                if(group[i].city[k]==t)
                {
                    flag=0;
                    break;
                }
            }
            if(flag)
            {
                group[i].city[j]=t;
                j++;
            }
        }
    }
    //打印种群基因
    printf("初始的种群\n");
    for(i=0;i<num;i++)
    {
        for(j=0;j<cities;j++)
        printf("%4d",group[i].city[j]);
        printf("\n");
    }
}
//评价函数,找出最优染色体
void pingjia()
{
    int i,j;
    int n1,n2;
    int sumdistance,biggestsum=0;
    double biggestp=0;
    for(i=0;i<num;i++)
    {
        sumdistance=0;
        for(j=1;j<cities;j++)
        {
            n1=group[i].city[j-1];
```

```
94.             n2=group[i].city[j];
95.             sumdistance+=distance[n1][n2];
96.         }
97.         group[i].adapt=sumdistance; //每条染色体的路径总和
98.         biggestsum+=sumdistance; //种群的总路径
99.     }
100.        //计算染色体的幸存能力,路径越短生存概率越大
101.        for(i=0;i<num;i++)
102.        {
103.            group[i].p=1-(double)group[i].adapt/(double)biggestsum;

104.            biggestp+=group[i].p;
105.        }
106.        for(i=0;i<num;i++)
107.        group[i].p=group[i].p/biggestp;  //在种群中的幸存概率,总和为
1
108.        //求最佳路径
109.        bestsolution=0;
110.        for(i=0;i<num;i++)
111.        if(group[i].p>group[bestsolution].p)
112.        bestsolution=i;
113.        //打印适应度
114.        for(i=0;i<num;i++)
115.        printf("染色体%d 的路径之和与生存概率分别
为%4d  %.4f\n",i,group[i].adapt,group[i].p);
116.        printf("当前种群的最优染色体是%d 号染色体\n",bestsolution);
117.    }
118.    //选择
119.    void xuanze()
120.    {
121.        int i,j,temp;
122.        double gradient[num];//梯度概率
123.        double xuanze[num];//选择染色体的随机概率
124.        int xuan[num];//选择了的染色体
125.        //初始化梯度概率
126.        for(i=0;i<num;i++)
127.        {
128.            gradient[i]=0.0;
129.            xuanze[i]=0.0;
130.        }
131.        gradient[0]=group[0].p;
132.        for(i=1;i<num;i++)
133.        gradient[i]=gradient[i-1]+group[i].p;
134.        srand((unsigned)time(NULL));
135.        //随机产生染色体的存活概率
```

```
        for(i=0;i<num;i++)
        {
            xuanze[i]=(rand()%100);
            xuanze[i]/=100;
        }
        //选择能生存的染色体
        for(i=0;i<num;i++)
        {
            for(j=0;j<num;j++)
            {
                if(xuanze[i]<gradient[j])
                {
                    xuan[i]=j; //第 i 个位置存放第 j 个染色体
                    break;
                }
            }
        }
        //拷贝种群
        for(i=0;i<num;i++)
        {
            grouptemp[i].adapt=group[i].adapt;
            grouptemp[i].p=group[i].p;
            for(j=0;j<cities;j++)
            grouptemp[i].city[j]=group[i].city[j];
        }
        //数据更新
        for(i=0;i<num;i++)
        {
            temp=xuan[i];
            group[i].adapt=grouptemp[temp].adapt;
            group[i].p=grouptemp[temp].p;
            for(j=0;j<cities;j++)
            group[i].city[j]=grouptemp[temp].city[j];
        }
        //用于测试
        /*
        printf("<------------------------------>\n");
        for(i=0;i<num;i++)
        {
            for(j=0;j<cities;j++)
            printf("%4d",group[i].city[j]);
            printf("\n");
            printf("染色体%d 的路径之和与生存概率分别
为%4d  %.4f\n",i,group[i].adapt,group[i].p);
        }
```

```
180.        */
181.    }
182.    //交配,对每个染色体产生交配概率,满足交配率的染色体进行交配
183.    void  jiaopei()
184.    {
185.        int i,j,k,kk;
186.        int t;//参与交配的染色体的个数
187.        int point1,point2,temp;//交配断点
188.        int pointnum;
189.        int temp1,temp2;
190.        int map1[cities],map2[cities];
191.        double jiaopeip[num];//染色体的交配概率
192.        int jiaopeiflag[num];//染色体的可交配情况
193.        int kkk,flag=0;
194.        //初始化
195.        for(i=0;i<num;i++)
196.        {
197.            jiaopeiflag[i]=0;
198.        }
199.        //随机产生交配概率
200.        srand((unsigned)time(NULL));
201.        for(i=0;i<num;i++)
202.        {
203.            jiaopeip[i]=(rand()%100);
204.            jiaopeip[i]/=100;
205.        }
206.        //确定可以交配的染色体
207.        t=0;
208.        for(i=0;i<num;i++)
209.        {
210.            if(jiaopeip[i]<pc)
211.            {
212.                jiaopeiflag[i]=1;
213.                t++;
214.            }
215.        }
216.        t=t/2*2;//t 必须为偶数
217.        //产生 t/2 个 0-9 交配断点
218.        srand((unsigned)time(NULL));
219.        temp1=0;
220.        //temp1 号染色体和 temp2 染色体交配
221.        for(i=0;i<t/2;i++)
222.        {
223.            point1=rand()%cities;//交配点 1
224.            point2=rand()%cities;//交配点 2
```

```
225.            //选出一个需要交配的染色体 1
226.            for(j=temp1;j<num;j++)
227.            {
228.                if(jiaopeiflag[j]==1)
229.                {
230.                    temp1=j;
231.                    break;
232.                }
233.            }
234.            //选出另一个需要交配的染色体 2 与 1 交配
235.            for(j=temp1+1;j<num;j++)
236.            {
237.                if(jiaopeiflag[j]==1)
238.                {
239.                    temp2=j;
240.                    break;
241.                }
242.            }
243.            //进行基因交配
244.            if(point1>point2) //保证 point1<=point2
245.            {
246.                temp=point1;
247.                point1=point2;
248.                point2=temp;
249.            }
250.            //初始化
251.            memset(map1,-1,sizeof(map1));
252.            memset(map2,-1,sizeof(map2));
253.            //断点之间的基因产生映射
254.            for(k=point1;k<=point2;k++)
255.            {
256.                map1[group[temp1].city[k]]=group[temp2].city[k];
257.                map2[group[temp2].city[k]]=group[temp1].city[k];
258.            }
259.            //断点两边的基因互换
260.            for(k=0;k<point1;k++)
261.            {
262.                temp=group[temp1].city[k];
263.                group[temp1].city[k]=group[temp2].city[k];
264.                group[temp2].city[k]=temp;
265.            }
266.            for(k=point2+1;k<cities;k++)
267.            {
268.                temp=group[temp1].city[k];
269.                group[temp1].city[k]=group[temp2].city[k];
```

```
                group[temp2].city[k]=temp;
            }
            printf("处理冲突----------------------\n");
            //处理染色体 1 产生的冲突基因
            for(k=0;k<point1;k++)
            {
                for(kk=point1;kk<=point2;kk++)
                {
                    if(group[temp1].city[k]==group[temp1].city[kk])

                    {
                        group[temp1].city[k]=map1[group[temp1].city
[k]];
                        //find
                        for(kkk=point1;kkk<=point2;kkk++)
                        {
                            if(group[temp1].city[k]==group[temp1].c
ity[kkk])
                            {
                                flag=1;
                                break;
                            }
                        }
                        if(flag==1)
                        {
                            kk=point1-1;
                            flag=0;
                        }
                        else
                        {
                            flag=0;
                            break;
                        }
                    }
                }

            }
            for(k=point2+1;k<cities;k++)
            {
                for(kk=point1;kk<=point2;kk++)
                {
                    if(group[temp1].city[k]==group[temp1].city[kk])

                    {
```

```
                    group[temp1].city[k]=map1[group[temp1].city[k]];
                    //find
                    for(kkk=point1;kkk<=point2;kkk++)
                    {
                        if(group[temp1].city[k]==group[temp1].city[kkk])
                        {
                            flag=1;
                            break;
                        }
                    }
                    if(flag==1)
                    {
                        kk=point1-1;
                        flag=0;
                    }
                    else
                    {
                        flag=0;
                        break;
                    }
                }
            }
        }
        //处理 2 染色体产生的冲突基因
        for(k=0;k<point1;k++)
        {
            for(kk=point1;kk<=point2;kk++)
            {
                if(group[temp2].city[k]==group[temp2].city[kk])

                {
                    group[temp2].city[k]=map2[group[temp2].city[k]];
                    //find
                    for(kkk=point1;kkk<=point2;kkk++)
                    {
                        if(group[temp2].city[k]==group[temp2].city[kkk])
                        {
                            flag=1;
                            break;
                        }
                    }
```

```
                        if(flag==1)
                        {
                            kk=point1-1;
                            flag=0;
                        }
                        else
                        {
                            flag=0;
                            break;
                        }
                    }
                }
            }
            for(k=point2+1;k<cities;k++)
            {
                for(kk=point1;kk<=point2;kk++)
                {
                    if(group[temp2].city[k]==group[temp2].city[kk])

                    {
                        group[temp2].city[k]=map2[group[temp2].city
[k]];
                        //find
                        for(kkk=point1;kkk<=point2;kkk++)
                        {
                            if(group[temp2].city[k]==group[temp2].c
ity[kkk])
                            {
                                flag=1;
                                break;
                            }
                        }
                        if(flag==1)
                        {
                            kk=point1-1;
                            flag=0;
                        }
                        else
                        {
                            flag=0;
                            break;
                        }
                    }
                }
            }
```

```
392.            temp1=temp2+1;
393.        }
394.    }
395.    //变异
396.    void bianyi()
397.    {
398.        int i,j;
399.        int t;
400.        int temp1,temp2,point;
401.        double bianyip[num]; //染色体的变异概率
402.        int bianyiflag[num];//染色体的变异情况
403.        for(i=0;i<num;i++)//初始化
404.        bianyiflag[i]=0;
405.        //随机产生变异概率
406.        srand((unsigned)time(NULL));
407.        for(i=0;i<num;i++)
408.        {
409.            bianyip[i]=(rand()%100);
410.            bianyip[i]/=100;
411.        }
412.        //确定可以变异的染色体
413.        t=0;
414.        for(i=0;i<num;i++)
415.        {
416.            if(bianyip[i]<pm)
417.            {
418.                bianyiflag[i]=1;
419.                t++;
420.            }
421.        }
422.        //变异操作,即交换染色体的两个节点
423.        srand((unsigned)time(NULL));
424.        for(i=0;i<num;i++)
425.        {
426.            if(bianyiflag[i]==1)
427.            {
428.                temp1=rand()%10;
429.                temp2=rand()%10;
430.                point=group[i].city[temp1];
431.                group[i].city[temp1]=group[i].city[temp2];
432.                group[i].city[temp2]=point;
433.            }
434.        }
435.    }
436.    int main()
```

```
437.    {
438.        int i,j,t;
439.        init();
440.        groupproduce();
441.        //初始种群评价
442.        pingjia();
443.        t=0;
444.        while(t++<MAXX)
445.        {
446.             xuanze();
447.             jiaopei();
448.             bianyi();
449.             pingjia();
450.        }
451.        //最终种群的评价
452.        printf("\n 输出最终的种群评价\n");
453.        for(i=0;i<num;i++)
454.        {
455.            for(j=0;j<cities;j++)
456.            {
457.                printf("%4d",group[i].city[j]);
458.            }
459.            printf("  adapt:%4d, p:%.4f\n",group[i].adapt,group[i].
p);
460.        }
461.        printf("最优解为%d 号染色体\n",bestsolution);
462.        return 0;
463.    }
```