昆明理工大学信息工程与自动化学院学生上机报告

（ 2021 —2022 学年 第 二 学期 ）

课程名称：人工智能　　上机教室：445　　2022 年 4 月 12 日

年级、专业、班	计科 202	学号	202010401137	姓名	丁紫嫣	成绩	
上机项目名称	智能算法				指导教师	钱谦	
师评语					教师签名： 年　月　日		

一、实验目的

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。

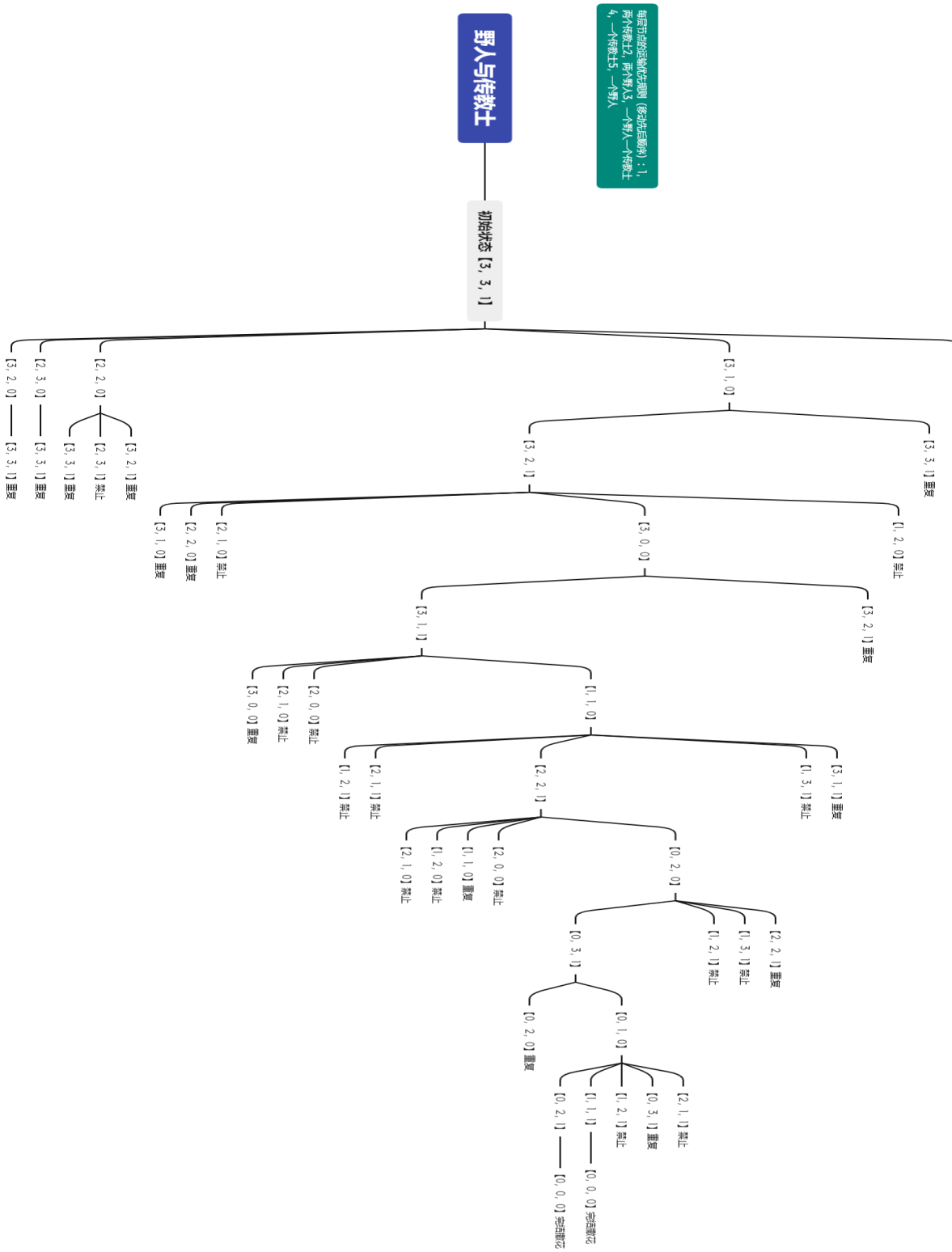
二、 实验原理及基本技术路线图（方框原理图）

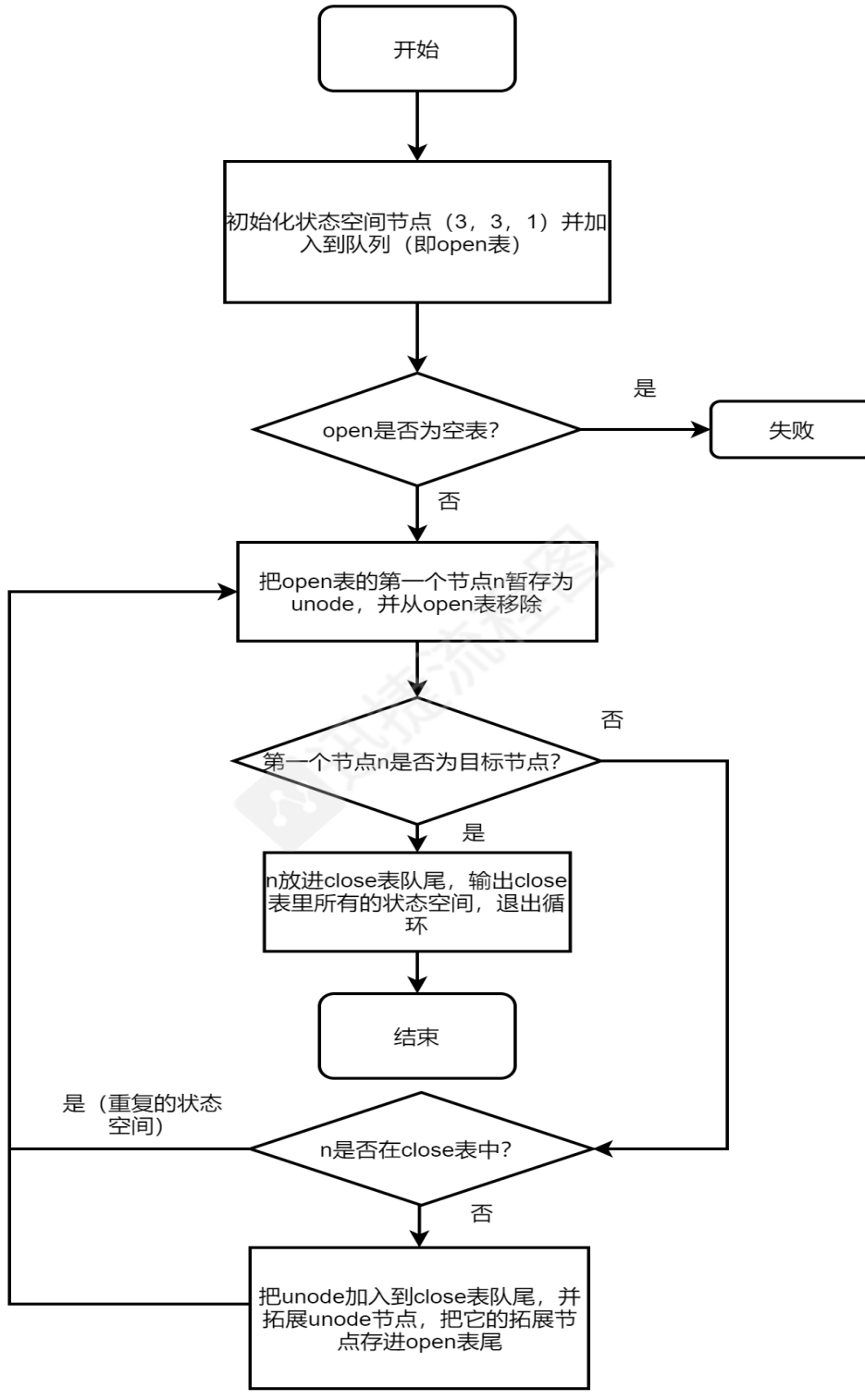
野人与传教士渡河问题：3 个野人与 3 个传教士打算乘一条船到对岸去，该船一次最多能运 2 个人，在任何时候野人人数超过传教士人数，野人就会把传教士吃掉，如何用这条船把所有人安全的送到对岸？在实现基本程序的基础上实现 N 个野人与 N 个传教士问题的所有解的求解，N 由用户输入。

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。

请加上程序功能结构图、流程图、数据结构定义、主要变量的说明、函数的说明等

状态空间图搜寻最优解：


每层节点的运输优先顺序（移动先后顺序）：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
野人与传教士
初始状态【3，3，1】
【1，3，0】禁止
【3，3，1】重复
【3，1，0】
【3，2，1】
【1，2，0】禁止
【3，0，0】
【3，2，1】重复
【3，1，1】
【1，1，0】
【3，1，1】重复
【1，3，1】禁止
【2，2，1】
【0，2，0】
【2，2，1】重复
【1，3，1】禁止
【1，2，1】禁止
【0，3，1】
【0，1，0】
【2，1，1】禁止
【0，3，1】重复
【1，2，1】禁止
【1，1，1】
【0，0，0】完结撒花
【0，2，1】
【0，0，0】完结撒花
【0，2，0】重复
【2，0，0】禁止
【1，1，0】重复
【1，2，0】禁止
【2，1，0】禁止
【2，1，1】禁止
【1，2，1】禁止
【2，0，0】禁止
【2，1，0】禁止
【3，0，0】重复
【3，1，0】重复
【2，2，0】禁止
【2，1，0】禁止
【2，2，0】
【3，2，1】重复
【3，3，1】重复
【2，3，0】
【3，3，1】重复
【3，2，0】
【3，3，1】重复


开始
初始化状态空间节点（3，3，1）并加入到队列（即open表）
open是否为空表?
是
失败
否
把open表的第一个节点n暂存为unode，并从open表移除
第一个节点n是否为目标节点?
否
是
n放进close表队尾，输出close表里所有的状态空间，退出循环
结束
是（重复的状态空间）
n是否在close表中?
否
把unode加入到close表队尾，并拓展unode节点，把它的拓展节点存进open表尾

每层节点的运输优先规则（移动先后顺序）：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

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：

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
```

三、所用仪器、材料（设备名称、型号、规格等）。

计算机一台

四、实验方法、步骤

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]

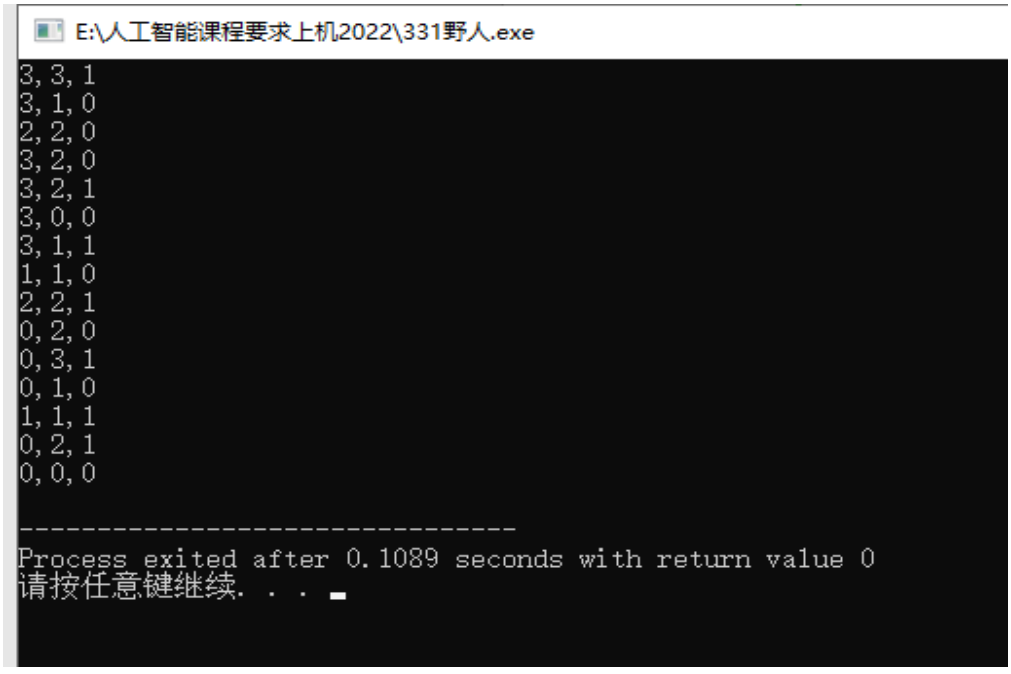
宽度优先搜索 3 野人 3 传教士代码如下（深度优先搜索需要把 93 行注释掉并把 92 行的注释改成代码）：

```
1. #include <iostream>
2. #include <vector>
3. #include <list>
4. using namespace std;
5.
6. 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++)
    {
        if(tNode==closed[i])
            return true;
    }
    if(i==closed.size())
        return false;
}
void ExpandNode(MCNode tNode,int b,list<MCNode> &open)
{
    MCNode node[5];//应用 5 条规则集生成新结点
    if(b==1)
    {
        for(int i=0;i<5;i++)
            node[i].b=0;
        node[0].m=tNode.m-2;
        node[0].c=tNode.c;
        node[1].m=tNode.m;
        node[1].c=tNode.c-2;
        node[2].m=tNode.m-1;
        node[2].c=tNode.c-1;
        node[3].m=tNode.m-1;
        node[3].c=tNode.c;
        node[4].m=tNode.m;
        node[4].c=tNode.c-1;
    }
    else
    {
        for(int i=0;i<5;i++)
            node[i].b=1;
        node[0].m=tNode.m+2;
        node[0].c=tNode.c;
        node[1].m=tNode.m;
        node[1].c=tNode.c+2;
        node[2].m=tNode.m+1;
        node[2].c=tNode.c+1;
        node[3].m=tNode.m+1;
        node[3].c=tNode.c;
        node[4].m=tNode.m;
        node[4].c=tNode.c+1;
    }
    for(int i=0;i<5;i++)
        if(IsLegal(node[i])&&!IsClosed(node[i]))
```

```
            //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
            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<<","<<closed[i].b<<endl;
                break;
            }
            if(!IsClosed(unode))
            {
                closed.push_back(unode);
                ExpandNode(unode,unode.b,open);
            }
        }
        return 0;
    }
```



可以发现答案与我们所作的状态空间图一致。

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。

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：

States* Start2Destination(States* pre,int x1, int x2)//用于返回 now 当前状态节点,适用于从开始到目的地的情况(x1 是船上牧师人数，x2 是船上野人人数)

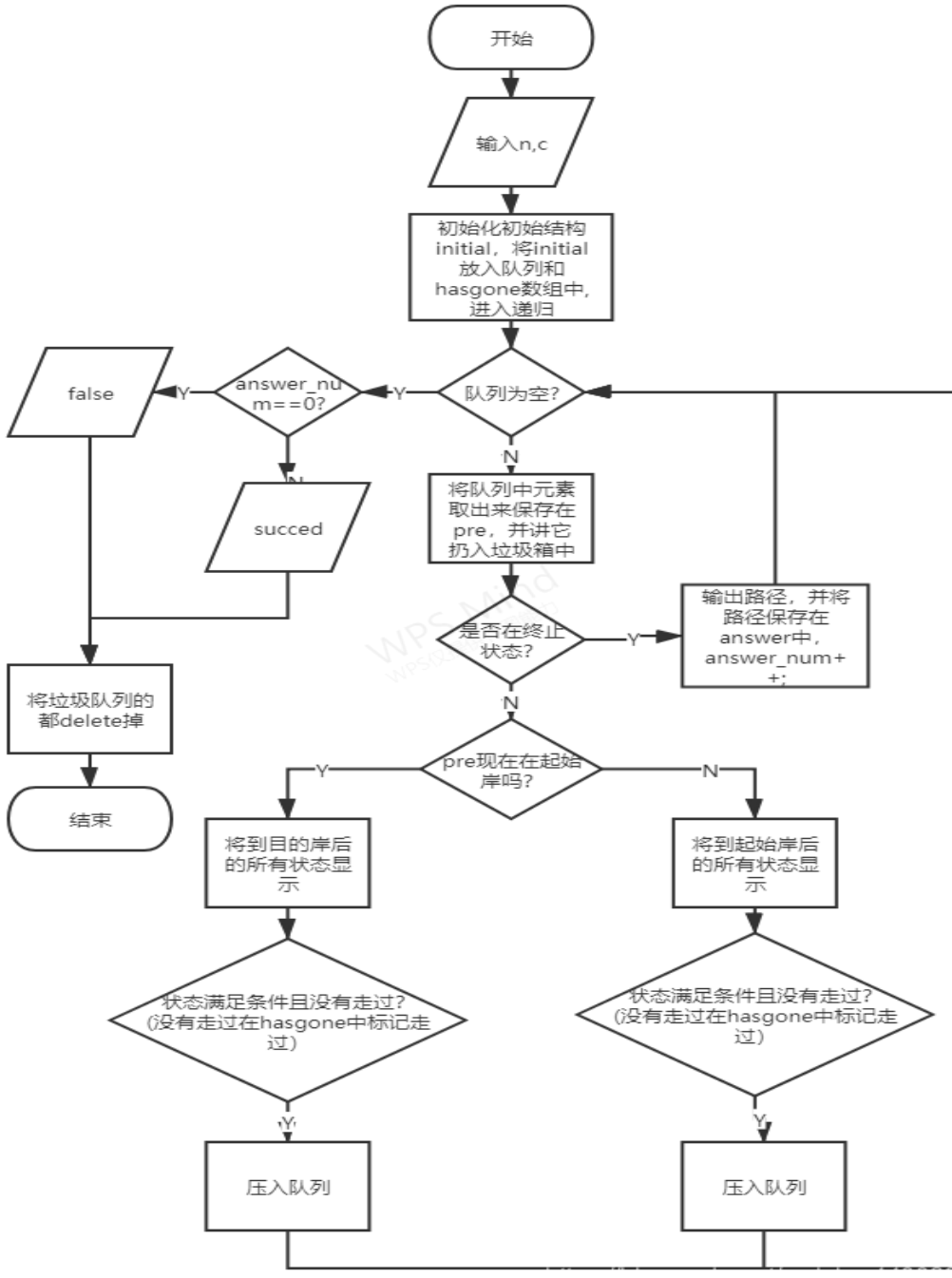
States* Destination2Start(States* pre,int x1, int x2)//用于返回 now 当前状态节点，适用于从目的地到开始地的情况(x1 是船上牧师人数，x2 是船上野人人数)

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

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

void　Nextstep() //处理出入队列逻辑的函数

程序框图：



代码如下：

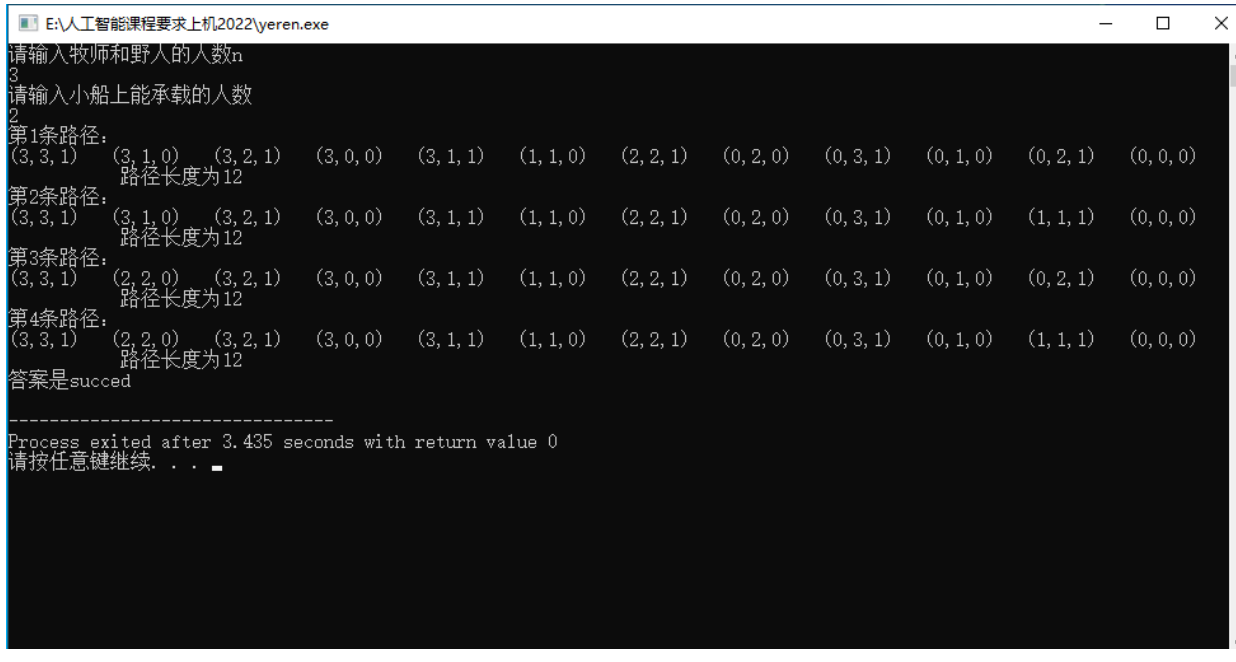
```
1. //基于广度优先搜索求解牧师和野人的问题
2. #include <iostream>
3. #include<queue>
4. //#include<Query.h>
5. using namespace std;
6.
7. struct States
8. {
9.     int X1;//起始岸上的牧师人数
10.     int X2;//起始岸上的野人人数
11.     int X3;//小船现在位置(1 表示起始岸，0 表示目的岸)
12.     int tree_depth = 0;//该状态第一次出现的最低层
13.     States * pre;//指向前驱的指针
14. };
15. States * hasgone[1000];//已经走过的状态,用于避免以后重复
16. int num = 0;//hasgone 的数量
17. int n = 0;//牧师和野人的总人数
18. int c = 0;//小船上可承载人数
19. queue<States *> que;//队列，用于进行广度优先搜索
20. //保存路径，以及它对应的船来回次数
21. States *answer[1000];
22. int num_answer = 0;
23. int step_num[1000];//路径对应的步数
24.
25. //int tree_depth = 0;
26. queue<States*> rubbish;//垃圾队列,用来存放最后运算完未 delete 的数据
27.
28.
29. 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)
30. {
31.     int Start_Priests= pre->X1 - x1;//现在岸上的牧师人数
32.     int Start_savages= pre->X2 - x2;//现在岸上的野人人数
33.     int End_Priests = n-pre->X1 + x1;//现在目的地的牧师人数
34.     int End_savages = n-pre->X2 +x2;//现在目的地的野人人数
35.     if ((x1 >= x2||x1==0)&& (Start_Priests >= Start_savages|| Start_Priests==0) &
    &( End_Priests >= End_savages|| End_Priests==0))//符合要求
36.     {
37.         States * now = new States;
38.         now->pre = pre;
```

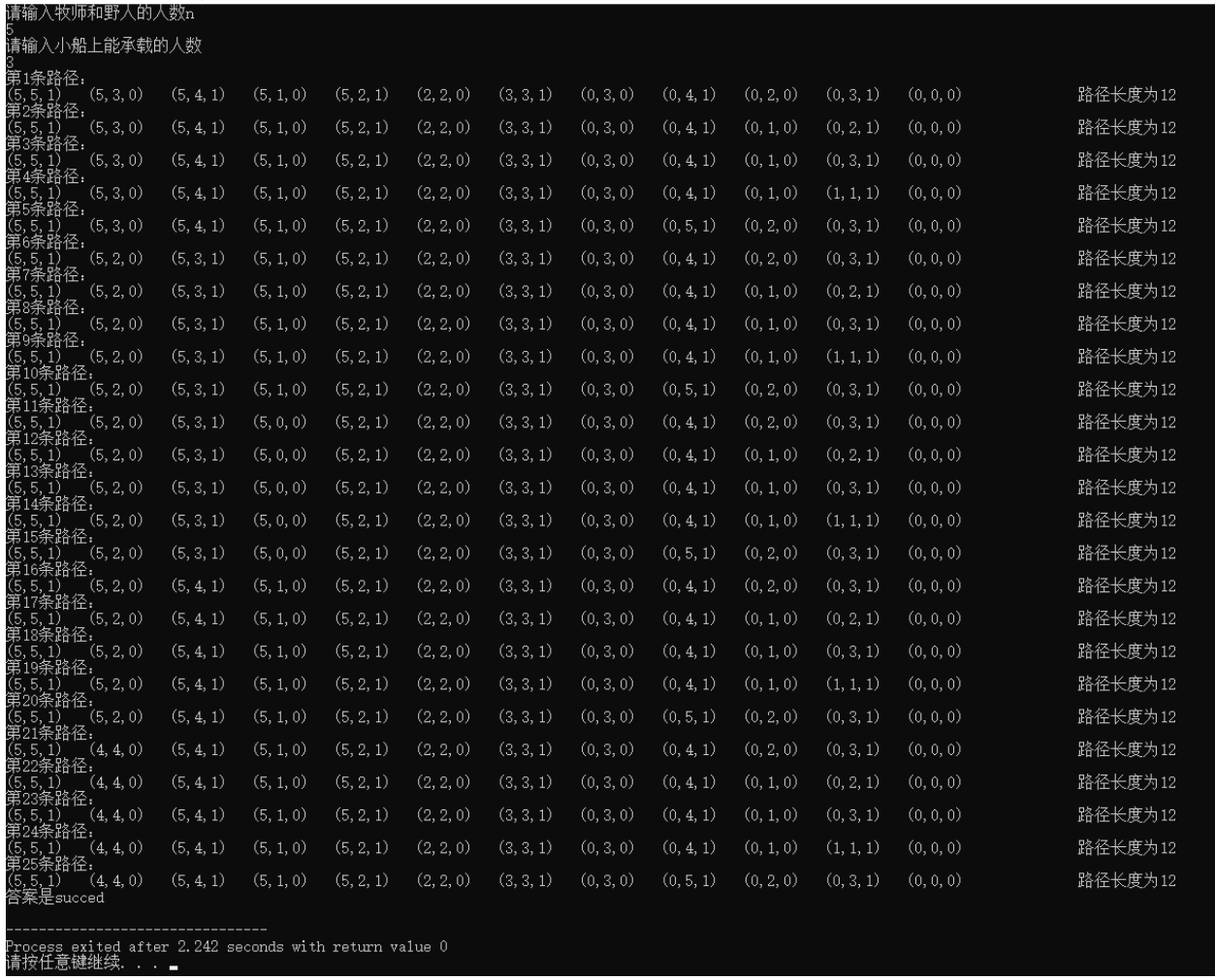
```
39.         now->X1 = Start_Priests;
40.         now->X2 = Start_savages;
41.         now->tree_depth = pre->tree_depth + 1;
42.         now->X3 = 0;
43.         return now;
44.     }
45.     return NULL;
46. }
47.
48. States* Destination2Start(States* pre,int x1, int x2)//从目的地到开始地(x1 是船上牧
    师人数，x2 是船上野人人数)
49. {
50.     int Start_Priests = pre->X1 +x1;//现在岸上的牧师人数
51.     int Start_savages = pre->X2 + x2;//现在岸上的野人人数
52.     int End_Priests = n - pre->X1 - x1;//现在目的地的牧师人数
53.     int End_savages = n - pre->X2 - x2;//现在目的地的野人人数
54.     if ((x1 >= x2 || x1 == 0) && (Start_Priests >= Start_savages || Start_Priests
     == 0) && (End_Priests >= End_savages || End_Priests == 0))//符合要求
55.     {
56.         States * now = new States;
57.         now->pre = pre;
58.         now->X1 = Start_Priests;
59.         now->X2 = Start_savages;
60.         now->tree_depth = pre->tree_depth + 1;
61.         now->X3 = 1;
62.         return now;
63.
64.     }
65.     return NULL;
66. }
67.
68. void showSolution(States * states, int time)
69. {
70.     if (states == NULL)
71.     {
72.         step_num[num_answer] = time;
73.         cout << endl;
74.     }
75.     else
76.     {
77.         showSolution(states->pre, ++time);
78.         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;
```

```
120.            int Max2 = pre->X2 > c ? c : pre->X2;
121.            for (int i = 0; i <= Max1; i++)
122.            {
123.                for (int j = 0; j <= Max2; j++)
124.                {
125.                    if (i + j <= c&&i+j>=1)
126.                    {
127.                        States* now=Start2Destination(pre,i,j);
128.                        if (now!=NULL&&!Hasgone(now))
129.                        {
130.                            que.push(now);
131.                        }
132.                    }
133.                }
134.            }
135.            Nextstep();
136.        }
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;
181.        initial->tree_depth = 0;
182.        initial->pre = NULL;
183.        que.push(initial);
184.        Hasgone(initial);
185.        Nextstep();
186.        if (num_answer == 0) cout << "答案是 false" << endl;
187.        else cout << "答案是 succed" << endl;
188.        delete_rubbish();
```





从输入 3 野人和传教士，船最多运输两人输出了四条路径来看，当求所有解和求路径最优解时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。

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。

如果依然要求最优解而非所有路径，盲目搜索容易造成资源浪费，此时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

代码如下：

```
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]
```

```
传教士和野人的人数（默认相同）：3
船的最大容量：2
搜索成功!
共生成节点数：14，共搜索节点数:12
过河方案如下：
      [M, C, B]
----> [3, 3, 1]
----> [2, 2, 0]
----> [3, 2, 1]
----> [3, 0, 0]
----> [3, 1, 1]
----> [1, 1, 0]
----> [2, 2, 1]
----> [0, 2, 0]
----> [0, 3, 1]
----> [0, 1, 0]
----> [0, 2, 1]
----> [0, 0, 0]

Process finished with exit code 0
```

```
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
```

遗传算法解 TSP 问题：

遗传算法的基本运算过程如下：

a)初始化：设置进化代数计数器 t=0，设置最大进化代数 T，随机生成 M 个个体作为初始群体 P(0)。

b)个体评价：计算群体 P(t)中各个个体的适应度。

c)选择运算:将选择算子作用于群体。选择的目的是把优化的个体直接遗传到下一代或通过配对交叉产生新的个体再遗传到下一代。选择操作是建立在群体中个体的适应度评估基础上的。

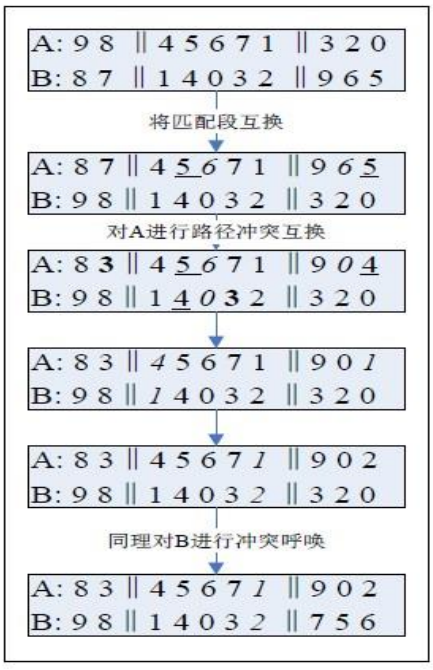
d)交叉运算：将交叉算子作用于群体。所谓交叉是指把两个父代个体的部分结构加以替换重组而生成新个体的操作。遗传算法中起核心作用的就是交叉算子。

e)变异运算：将变异算子作用于群体。即是对群体中的个体串的某些基因座上的基因值作变动。

群体 P(t)经过选择、交叉、变异运算之后得到下一代群体 P(t 1)。

f)终止条件判断:若 t=T,则以进化过程中所得到的具有最大适应度个体作为最优解输出，终止计算。

本题采用第三交叉策略：



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）

适应度函数（生存概率）：总路径越小，生存概率越高

我们这里以生存概率 **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

主要结构体：

```
struct group  //染色体的结构
{
                                    int city[cities];//城市的顺序
                                    int adapt;//适应度
                                    double p;//在种群中的幸存概率
}group[num],grouptemp[num];
```

主要的变量：

```
#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
```

主要函数：

```
void init()//随机产生 cities 个城市之间的相互距离，打印距离矩阵
void groupproduce()//随机产生初试群
void pingjia()//评价函数,找出最优染色体
void xuanze()//选择
void  jiaopei()//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
void bianyi()//变异
```

代码如下：

```
#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];
            n2=group[i].city[j];
            sumdistance+=distance[n1][n2];
        }
        group[i].adapt=sumdistance; //每条染色体的路径总和
        biggestsum+=sumdistance; //种群的总路径
    }
      //计算染色体的幸存能力,路径越短生存概率越大
      for(i=0;i<num;i++)
      {
          group[i].p=1-(double)group[i].adapt/(double)biggestsum;
          biggestp+=group[i].p;
      }
      for(i=0;i<num;i++)
      group[i].p=group[i].p/biggestp;  //在种群中的幸存概率,总和为 1
      //求最佳路径
      bestsolution=0;
      for(i=0;i<num;i++)
      if(group[i].p>group[bestsolution].p)
      bestsolution=i;
      //打印适应度
      for(i=0;i<num;i++)
      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);
      printf("当前种群的最优染色体是%d 号染色体\n",bestsolution);
  }
  //选择
  void xuanze()
  {
      int i,j,temp;
      double gradient[num];//梯度概率
      double xuanze[num];//选择染色体的随机概率
      int xuan[num];//选择了的染色体
      //初始化梯度概率
      for(i=0;i<num;i++)
      {
```

```
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
136.        for(i=0;i<num;i++)
137.        {
138.            xuanze[i]=(rand()%100);
139.            xuanze[i]/=100;
140.        }
141.        //选择能生存的染色体
142.        for(i=0;i<num;i++)
143.        {
144.            for(j=0;j<num;j++)
145.            {
146.                if(xuanze[i]<gradient[j])
147.                {
148.                    xuan[i]=j; //第 i 个位置存放第 j 个染色体
149.                    break;
150.                }
151.            }
152.        }
153.        //拷贝种群
154.        for(i=0;i<num;i++)
155.        {
156.            grouptemp[i].adapt=group[i].adapt;
157.            grouptemp[i].p=group[i].p;
158.            for(j=0;j<cities;j++)
159.            grouptemp[i].city[j]=group[i].city[j];
160.        }
161.        //数据更新
162.        for(i=0;i<num;i++)
163.        {
164.            temp=xuan[i];
165.            group[i].adapt=grouptemp[temp].adapt;
166.            group[i].p=grouptemp[temp].p;
167.            for(j=0;j<cities;j++)
168.            group[i].city[j]=grouptemp[temp].city[j];
169.        }
170.        //用于测试
```

```
171.        /*
172.        printf("<------------------------------->\n");
173.        for(i=0;i<num;i++)
174.        {
175.            for(j=0;j<cities;j++)
176.            printf("%4d",group[i].city[j]);
177.            printf("\n");
178.            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);
179.        }
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;
```

```
                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
```

```
342.                         for(kkk=point1;kkk<=point2;kkk++)
343.                         {
344.                             if(group[temp2].city[k]==group[temp2].city[kkk])
345.                             {
346.                                 flag=1;
347.                                 break;
348.                             }
349.                         }
350.                         if(flag==1)
351.                         {
352.                             kk=point1-1;
353.                             flag=0;
354.                         }
355.                         else
356.                         {
357.                             flag=0;
358.                             break;
359.                         }
360.                     }
361.                 }
362.             }
363.             for(k=point2+1;k<cities;k++)
364.             {
365.                 for(kk=point1;kk<=point2;kk++)
366.                 {
367.                     if(group[temp2].city[k]==group[temp2].city[kk])
368.                     {
369.                         group[temp2].city[k]=map2[group[temp2].city[k]];
370.                         //find
371.                         for(kkk=point1;kkk<=point2;kkk++)
372.                         {
373.                             if(group[temp2].city[k]==group[temp2].city[kkk])
374.                             {
375.                                 flag=1;
376.                                 break;
377.                             }
378.                         }
379.                         if(flag==1)
380.                         {
381.                             kk=point1-1;
382.                             flag=0;
383.                         }
384.                         else
```

```
                    {
                        flag=0;
                        break;
                    }
                }
            }
        }
        temp1=temp2+1;
    }
}
//变异
void bianyi()
{
    int i,j;
    int t;
    int temp1,temp2,point;
    double bianyip[num]; //染色体的变异概率
    int bianyiflag[num];//染色体的变异情况
    for(i=0;i<num;i++)//初始化
    bianyiflag[i]=0;
    //随机产生变异概率
    srand((unsigned)time(NULL));
    for(i=0;i<num;i++)
    {
        bianyip[i]=(rand()%100);
        bianyip[i]/=100;
    }
    //确定可以变异的染色体
    t=0;
    for(i=0;i<num;i++)
    {
        if(bianyip[i]<pm)
        {
            bianyiflag[i]=1;
            t++;
        }
    }
    //变异操作,即交换染色体的两个节点
    srand((unsigned)time(NULL));
    for(i=0;i<num;i++)
    {
        if(bianyiflag[i]==1)
        {
```

```
                temp1=rand()%10;
                temp2=rand()%10;
                point=group[i].city[temp1];
                group[i].city[temp1]=group[i].city[temp2];
                group[i].city[temp2]=point;
            }
        }
    }
    int main()
    {
        int i,j,t;
        init();
        groupproduce();
        //初始种群评价
        pingjia();
        t=0;
        while(t++<MAXX)
        {
             xuanze();
             jiaopei();
             bianyi();
             pingjia();
        }
        //最终种群的评价
        printf("\n 输出最终的种群评价\n");
        for(i=0;i<num;i++)
        {
            for(j=0;j<cities;j++)
            {
                printf("%4d",group[i].city[j]);
            }
            printf("  adapt:%4d, p:%.4f\n",group[i].adapt,group[i].p);
        }
        printf("最优解为%d 号染色体\n",bestsolution);
        return 0;
    }
```

运行结果：

```
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----------------------
```

```
输出最终的种群评价
   2   9   3   5   7   6   8   1   4   0  adapt: 299, p:0.1024
   2   9   3   5   7   6   8   1   4   0  adapt: 299, p:0.1024
   7   1   2   5   4   6   9   8   3   0  adapt: 457, p:0.0978
   2   9   3   5   7   6   8   1   4   0  adapt: 299, p:0.1024
   2   9   3   5   7   6   8   1   4   0  adapt: 299, p:0.1024
   2   9   3   5   7   6   8   1   4   0  adapt: 299, p:0.1024
   7   1   2   5   4   6   9   8   3   0  adapt: 457, p:0.0978
   7   1   2   5   4   6   3   8   0   9  adapt: 504, p:0.0965
   7   1   2   5   4   6   9   8   3   0  adapt: 457, p:0.0978
   7   1   2   5   4   6   9   8   3   0  adapt: 457, p:0.0978
最优解为0号染色体

--------------------------------
```

总结：

通过实例理解了深度优先算法，广度优先算法，A*算法等遍历寻找最优解的算法，也通过 TSP 问题学会了遗传算法的应用。对于深度优先搜索、广度优先搜索等盲目搜索算法，和启发性算法之 A*算法，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。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，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。

存在的问题：看得懂代码，也理解得了算法，但是自己写就不怎么写得出。主要还是代码量少了，而且队列和链表还有图不怎么熟练。

收获：自从转到计算机专业因为课太满还没修数据结构，上学期直接学了算法，为了写报告和看懂代码终于迫于压力基本彻底自学完了数据结构，知道了 C++模板库的存在但是还不怎么会用，并且决定以后写算法题多试着用 C++的 STL 模板库和 Python。

应该下去改进的方面：希望自己写出的代码方便简明，能让人看懂为主。另外应该多加尝试其他方法，比如说遗传算法解 TSP 问题里面的交叉函数使用的是交叉方法三，那么如果是交叉方法二，交叉方法一该怎么写？遗传算法在数学建模里使用多次，但是却还没有写出能够解决和贴合实际问题的代码，基本全部依靠 matlab 工具箱，我应该怎样把遗传算法模板化，写出符合应用功能的代码？应加强代码量的练习，从易到难地刷题，加深对算法的深度挖掘，以便研究生复试有话可说。