**中山大学计算机学院**

**人工智能**

**本科生实验报告**

**（2025学年春季学期）**

课程名称：Artificial Intelligence

|  |  |  |  |
| --- | --- | --- | --- |
| 教学班级 |  | 专业（方向） |  |
| 学号 |  | 姓名 |  |

# 实验题目

**（1）启发式搜索解决15-Puzzle问题**

利用A\*算法和IDA\*算法解决15-Puzzle问题, 可自定义启发式函数. Puzzle问题的输入数据类型为二维嵌套list,空位置用0表示.输出移动数字方块的次序.

**（2）遗传算法解决TSP问题**

编写类GeneticAlgTSP来使用遗传算法来解决TSP问题,并分析算法性能. 类至少需包含以下方法:

* 构造函数 \_\_init\_\_(),输入为TSP数据集文件名filename,数据类型 str. 在构造函数中读取该文件中的数据,存储到类成员self.cities中.同时初始化种群,存储到类成员self.population中.
* 求解方法 iterate(),输入为算法迭代的轮数 num\_iterations.基于当前种群 self.population进行迭代,返回迭代后种群中的一个较优解,格式为城市编号的排列.
* 在类中编写其他方法以方便编写并分析遗传算法的性能.为了更好地分析遗传算法的性能, 应该以不同的初始随机种子或用不同的参数(例如种群数量, 变异概率等)多次运行算法, 这些需要在实验报告中呈现.

# 实验内容

1. 算法原理
2. **启发式搜索**

评价函数的定义如下：

其中表示从初始节点到达节点的路劲成本，从节点到目标节点的启发式估计值

值得注意的是启发式函数**可采纳性意味着最优性，**在设计时往往需要考虑

* 算法：在深度优先搜索(BFS)的基础上，每次扩展探索评价函数最大的节点
* 算法：在迭代加深搜索算法的基础上，每次深度优先搜索的界限为上次迭代的评价函数

**(2)GA遗传算法：**借鉴生物界自然选择和遗传机制的一种随机搜索算法，通过模拟自然界生物进化的原理：遗传和变异、物竞天择、适者生存，来设计和实现GA算法的主要步骤编码、初代种群、解码、选择、交叉、变异几个步骤，随着迭代的次数增加，选择出最优个体

1. 伪代码

(1)算法

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Algorithm 1: A\_star** |  |  |  |  |  |  |  |
| **Input**: |  |  |  |  |  |  |  |
| **Return**: search\_path |  |  |  |  |  |  |  |
| Initialize: frontier ← [] // open表，存储待探索节点 |  |  |  |  |  |  |  |
| close ← None // close表，储存已探索节点 |  |  |  |  |  |  |  |
| while frontier not empty do |  |  |  |  |  |  |  |
| // 从开放表选取f值最小的节点 |  |  |  |  |  |  |  |
| s ← frontier.pop() with the lowest f(s) |  |  |  |  |  |  |  |
| // 将s加入close表 |  |  |  |  |  |  |  |
| close ← s |  |  |  |  |  |  |  |
| // 遍历s的所有下一个状态 |  |  |  |  |  |  |  |
| for s' in next\_states do |  |  |  |  |  |  |  |
| // 如果s'不在close表里，把s'加入到frontier表中 |  |  |  |  |  |  |  |
| if s' not in close then |  |  |  |  |  |  |  |
| frontier ← s' |  |  |  |  |  |  |  |
| end if |  |  |  |  |  |  |  |
| // 如果s'是目标，那么返回搜索路径 |  |  |  |  |  |  |  |
| if s' == target then |  |  |  |  |  |  |  |
| return search\_path |  |  |  |  |  |  |  |
| end if |  |  |  |  |  |  |  |
| end for |  |  |  |  |  |  |  |
| end while |  |  |  |  |  |  |  |
| return None |  |  |  |  |  |  |  |

(2)算法

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Algorithm 2: IDA\_star** |  |  |  |  |  |  |  |
| **Input**: |  |  |  |  |  |  |  |
| **Return**: search\_path |  |  |  |  |  |  |  |
| Initialize: bound ← Heurisic() + g() // 初始bound值 |  |  |  |  |  |  |  |
| while True do |  |  |  |  |  |  |  |
| initialize: close ← None |  |  |  |  |  |  |  |
| // 调用dfs获得最小花费 |  |  |  |  |  |  |  |
| min\_cost ← dfs() |  |  |  |  |  |  |  |
| // 如果最小花费为-1 表示已经找到目标，返回搜索路径 |  |  |  |  |  |  |  |
| if min\_cost == -1 then |  |  |  |  |  |  |  |
| return search\_path |  |  |  |  |  |  |  |
| // 更新bound值为上一次迭代的最小花费 |  |  |  |  |  |  |  |
| bound ← min\_cost |  |  |  |  |  |  |  |
| end while |  |  |  |  |  |  |  |
| **return** None |  |  |  |  |  |  |  |
| // 辅助函数dfs |  |  |  |  |  |  |  |
| **function** dfs |  |  |  |  |  |  |  |
| **Input**: s,bound |  |  |  |  |  |  |  |
| **Return**: min\_cost |  |  |  |  |  |  |  |
| // 递归结束条件：如果当前状态f值大于上限bound返回f |  |  |  |  |  |  |  |
| if f(s) > bound then |  |  |  |  |  |  |  |
| return f(s) |  |  |  |  |  |  |  |
| end if |  |  |  |  |  |  |  |
| // 递归结束条件：如果找到目标，返回-1 表示结束 |  |  |  |  |  |  |  |
| if s == target then |  |  |  |  |  |  |  |
| return -1 |  |  |  |  |  |  |  |
| end if |  |  |  |  |  |  |  |
| // 遍历s的所有下一个状态 |  |  |  |  |  |  |  |
| for s' in next\_states do |  |  |  |  |  |  |  |
| if s' not in close then |  |  |  |  |  |  |  |
| close ← s' |  |  |  |  |  |  |  |
| // 递归调用dfs |  |  |  |  |  |  |  |
| cost ← dfs(s') |  |  |  |  |  |  |  |
| if cost == -1 then |  |  |  |  |  |  |  |
| return -1 |  |  |  |  |  |  |  |
| end if |  |  |  |  |  |  |  |
| // 返回cost和min\_cost的最小值 |  |  |  |  |  |  |  |
| if cost < min\_cost then |  |  |  |  |  |  |  |
| min\_cost ← cost |  |  |  |  |  |  |  |
| end if |  |  |  |  |  |  |  |
| end if |  |  |  |  |  |  |  |
| end for |  |  |  |  |  |  |  |
| return min\_cost |  |  |  |  |  |  |  |

(3)GA遗传算法

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Algorithm 3: TSP of GeneticAlgorithm** |  |  |  |  |  |  |  |  |
| **Input**: Coordinates of n cities |  |  |  |  |  |  |  |  |
| **Return**: best solution |  |  |  |  |  |  |  |  |
| Initialize: population\_size // 种群大小 |  |  |  |  |  |  |  |  |
| num\_iterations // 迭代次数 |  |  |  |  |  |  |  |  |
| mutation\_rate // 变异率 |  |  |  |  |  |  |  |  |
| P(0) ← encoding routinue |  |  |  |  |  |  |  |  |
| for t=0 to num\_interations do |  |  |  |  |  |  |  |  |
| for i=0 to population\_size do |  |  |  |  |  |  |  |  |
| // 依据fitness，从P(t)中选择两个个体作为亲本 |  |  |  |  |  |  |  |  |
| parent1,parent2 ← selction(P(t), 2) |  |  |  |  |  |  |  |  |
| // 两个亲本crossover产生两个子代 |  |  |  |  |  |  |  |  |
| childs ← crossover(parent1,parent2) |  |  |  |  |  |  |  |  |
| // 两个子代以一定的变异率产生变异 |  |  |  |  |  |  |  |  |
| childs ← mutation(childs,mutation\_rate) |  |  |  |  |  |  |  |  |
| // 产生的所有子代构成C(t) |  |  |  |  |  |  |  |  |
| C(t) ← childs |  |  |  |  |  |  |  |  |
| end for |  |  |  |  |  |  |  |  |
| // 根据适应度，从C(t)∪P(t)中选population\_size个构成下一代 |  |  |  |  |  |  |  |  |
| P(t+1) ← selction(C(t)∪P(t), population\_size) |  |  |  |  |  |  |  |  |
| end for |  |  |  |  |  |  |  |  |
| return best soulutin in last generation |  |  |  |  |  |  |  |  |

1. 关键代码展示

**启发式搜索：**

对于启发式搜索，首先我们需要设计我们的启发值函数，对于我们的15-puzzle问题本次实验有四种启发值函数，这四种启发值函数将会在后面的内容进行比较，这里效果最好的是第4个**线性冲突优化的曼哈顿距离启发值函数**，具体代码如下：

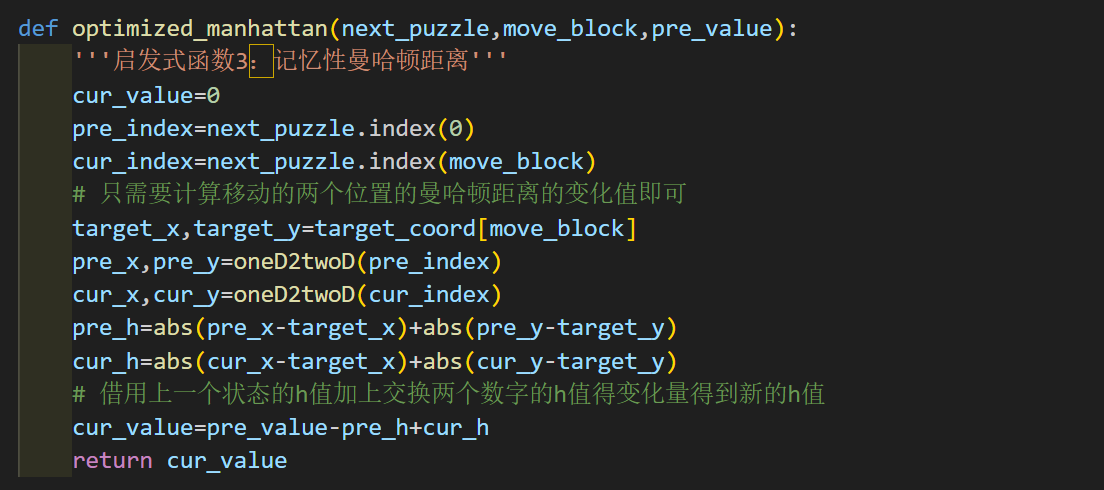
**①启发值函数1：错位方块数**



**②启发值函数2：曼哈顿距离**



**③启发值函数3：记忆性曼哈顿距离**

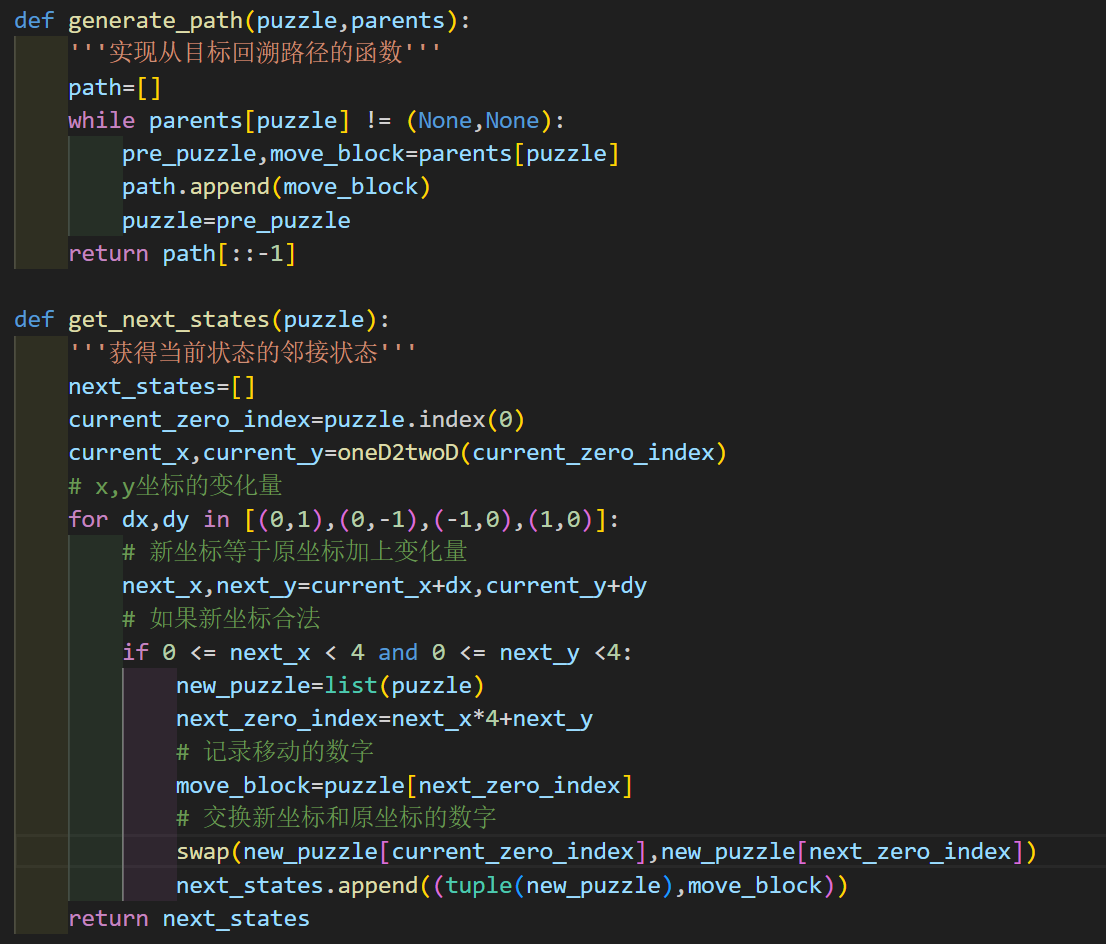


**④启发值函数4：线性冲突优化的曼哈顿距离**

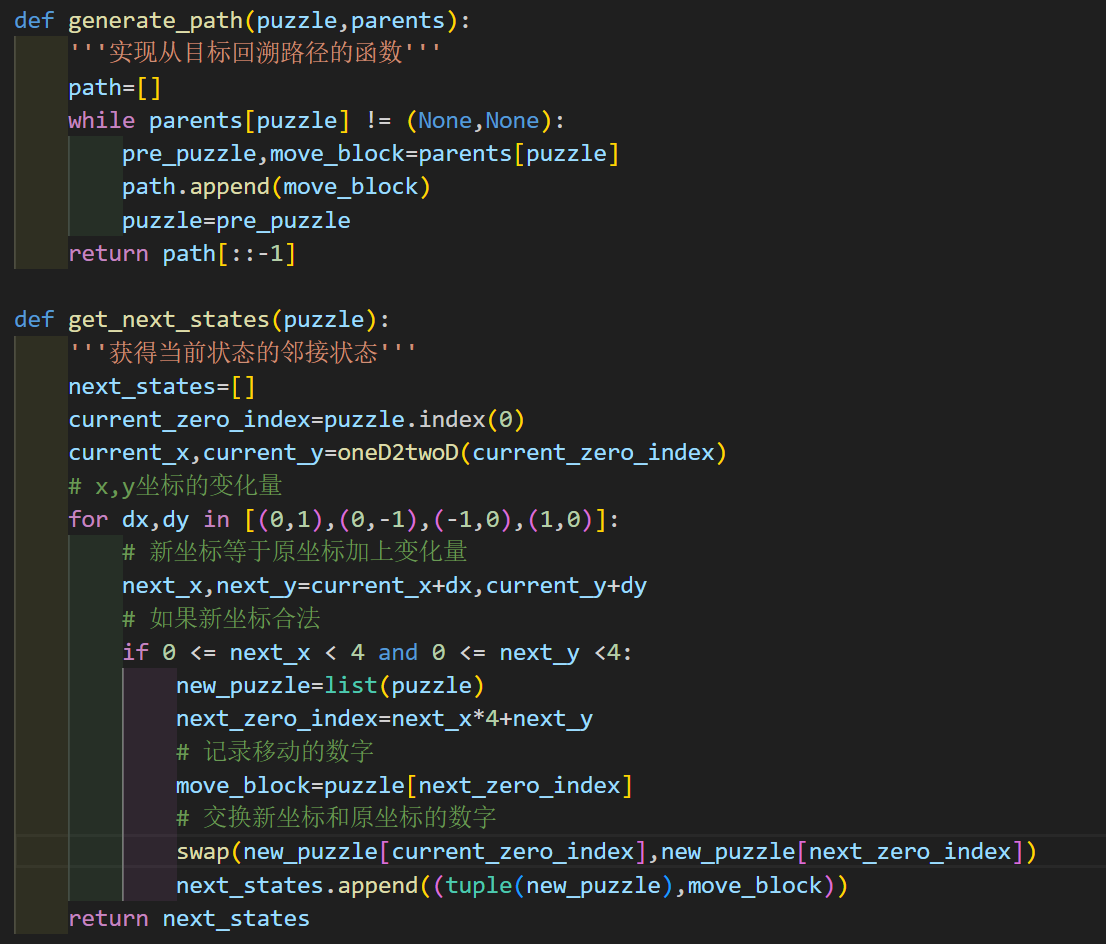


在设计好了启发值函数之后，我们需要设计两个辅助函数

**①get\_next\_states函数：实现获得当前状态的所有下一个状态**，具体就是当前0分别向上下左右数字交换得到所有的4个邻接状态，具体代码如下：

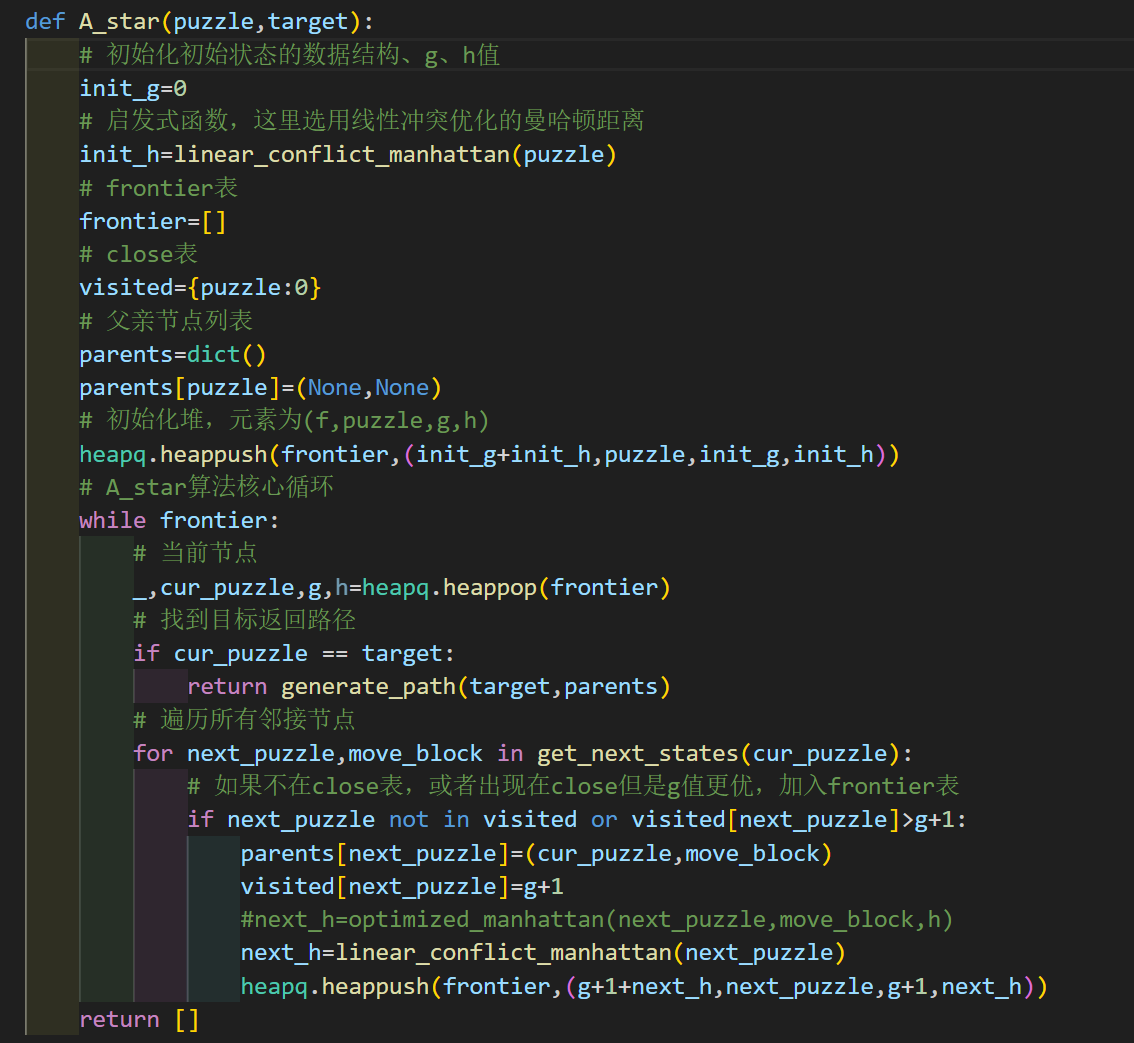


**②generate\_path函数：实现从目标值回溯到我们的初始状态已得到搜索路径**，具体实现就是通过记录当前节点的父亲节点parents字典，通过parents不断回溯，具体代码如下：



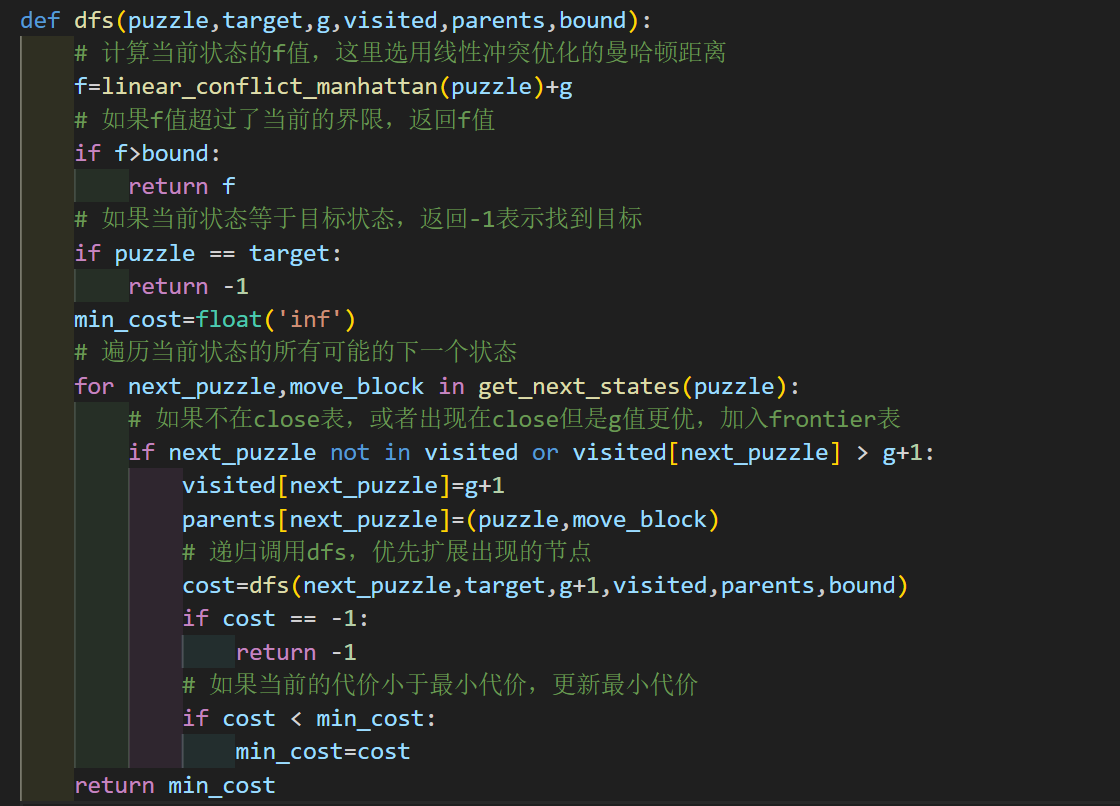
1. **算法**

**启发式宽度优先搜索算法**，有一个frontier表储存带探索的节点和一个close表储存已探索的节点，每次根据评价函数值最小的节点进行扩展，遍历该节点的所有邻接节点，如果邻接节点没在close表或者在close表出现但是实际代价值比已出现在close节点的值更小，那么就将其加入到frontier表，具体实现如下：



1. **算法**

**启发式的迭代加深搜索算法**，在迭代加深搜索算法的基础上，将每次迭代将上一次迭代搜索中大于上限bound的最小值作为本次迭代的上限(bound)值进行dfs，代码如下：



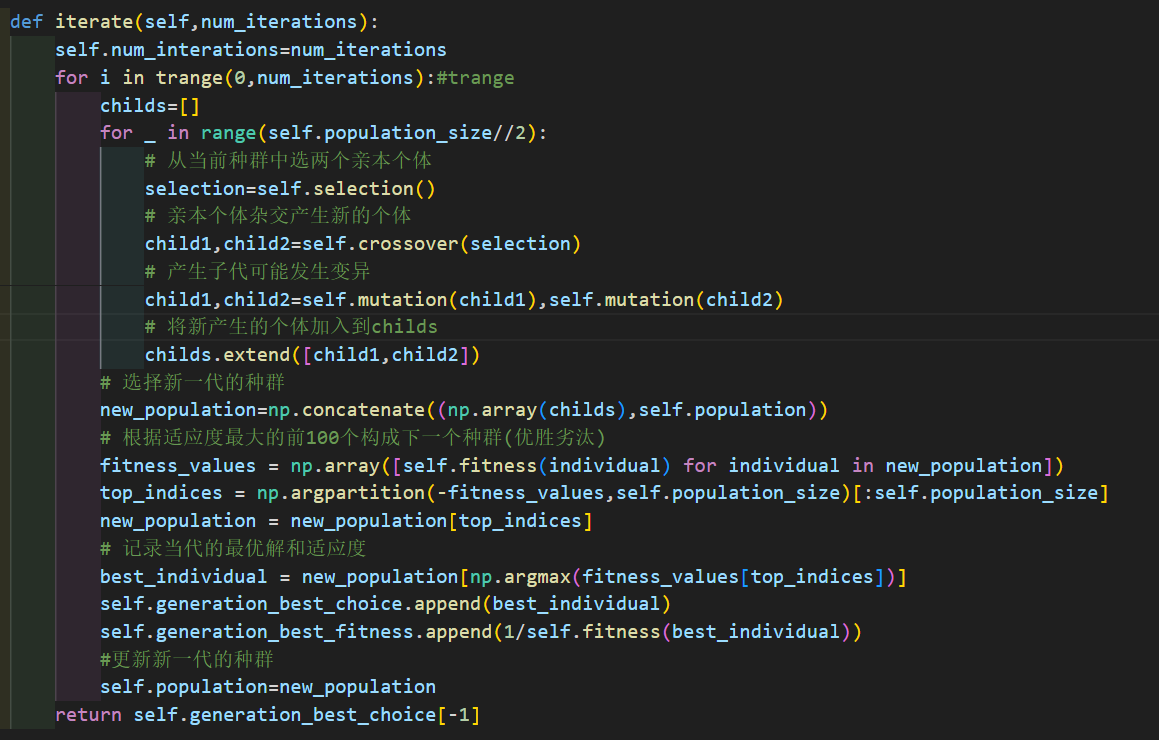


**GA遗传算法**

**代码框架以及核心函数的实现**：GA算法的核心步骤iterate函数：根据预先设定的迭代次数来设定循环次数，对于每一次迭代：

* 根据适应度fitness从当前种群中选择两个个体
* 选出的两个个体进行杂交crossover产生两个新的后代个体
* 两个后代个体以一定的概率发生变异mutation
* 根据适应度fitness从当前种群和产生的新个体中选population\_size个组成新一代的种群，用于下一次迭代

具体代码如下：

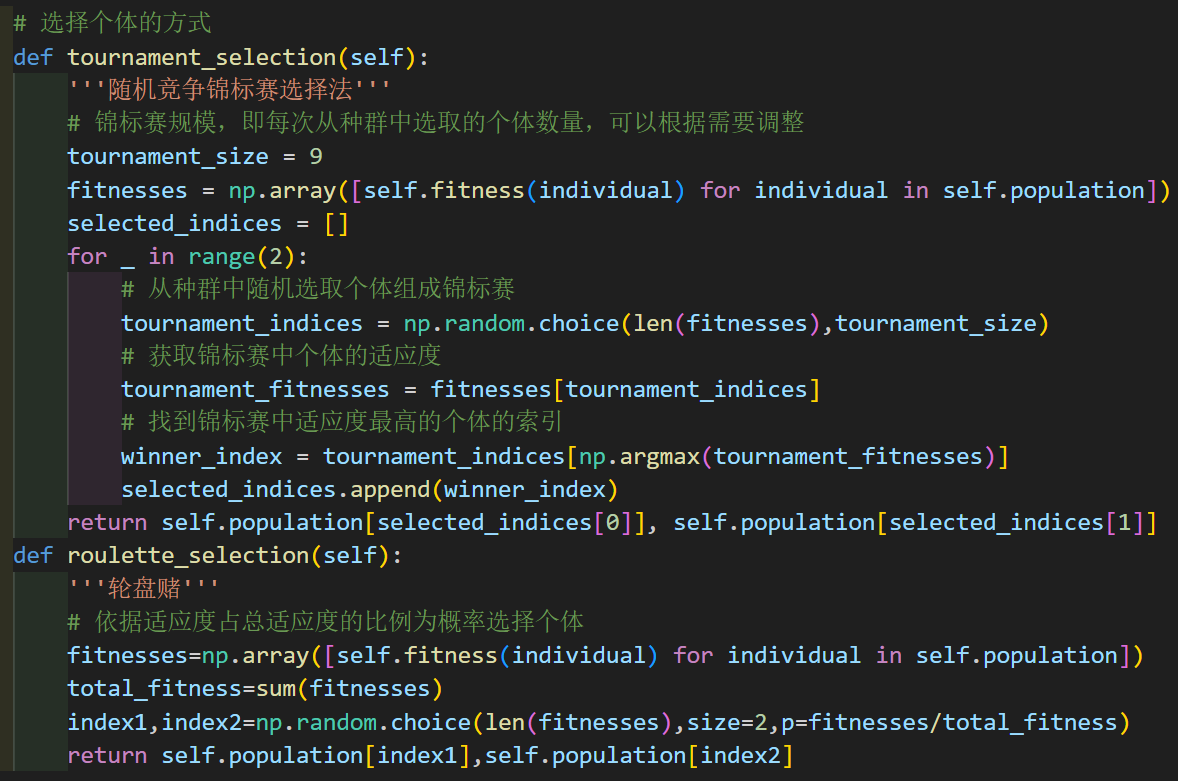


各个模块的具体实现方式：

**①selection函数：实现依据适应度fitness选择个体的**，选择方法有很多种，这里只实现了两种，一种是随机竞争的方式，另一种是使用轮盘赌的方式。

* 随机竞争方式指的是从种群中随机选择若干个体，从他们之中选择适应度最高的就是我们的优胜者被我们选中，循环的次数就是我们选择的个数
* 轮盘赌的方式指的是以适应度占总适应度的比例作为选择该个体的概率进行选择，实际上就是一个转轮盘的模型，轮盘上的面积表示选择概率即个体的适应度

具体代码如下：



②**crossover函数：实现两个亲代个体的交叉互换产生新的后代个体**，交叉互换的方式也有很多种，在本次实验中我主要实现了三种，分别是PMX、OX、PBX三种交叉互换方式：

* 部分匹配交叉PMX：挑选一个交叉区域，交换两个亲本该区间的基因，对于区间之外的发生冲突的基因，通过交叉区间的映射关系来进行修正，示例：

亲本1 [1,2,3,4,5,6,7,8,9] 子代1 [3,5,6,9,2,1,7,8,4]

亲本2 [5,4,6,9,2,1,7,8,3] 子代2 [2,9,3,4,,5,6,7,8,1]

* 顺序交叉OX：选择一个交叉区间，将一个亲本的基因直接复制到子代的相应区间，按照另一个亲本的基因顺序，依次把子代未出现的基因按顺序放到子代的相应位置。

示例：

亲本1 [1,2,3,4,5,6,7,8,9] 子代1 [3,4,6,9,2,1,5,7,8]

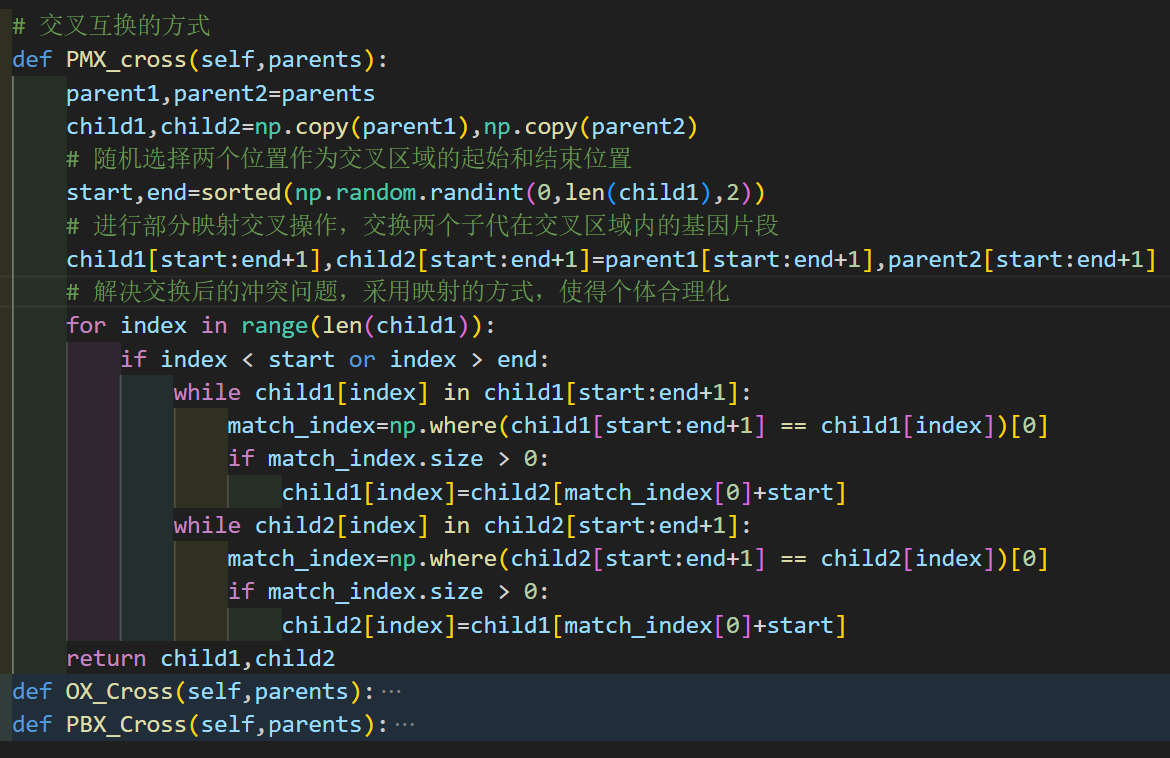
亲本2 [5,4,6,9,2,1,7,8,3] 子代2 [7,9,3,4,5,6,1,2,8]

* 基于位置的交叉PBX：随机选取一系列的位置，把一个亲本的对应位置的基因复制到子代的相应位置，按照另一个亲本的基因顺序，依次把子代未出现的基因按顺序放到子代的相应位置，示例：

亲本1 [1,2,3,4,5,6,7,8,9] 子代1 [4,2,3,1,5,6,7,8,9]

亲本2 [5,4,6,9,2,1,7,8,3] 子代2 [5,4,6,7,2,1,8,9,3]

具体代码如下：



**③muatation函数**：**实现子代个体的变异**，变异的当时同样的也有很多种实现方式，本次实验所实现的变异方式有4种，分别是倒置变异、插入变异、位移变异、交换变异

* 倒置变异：选择个体种的一段基因，将其倒置reserve
* 插入变异：选择一个位置的基因将其插入到另一个位置
* 位移变异：选择一段基因将其移动到另一个位置
* 交换变异：选择两个基因，交换两个基因

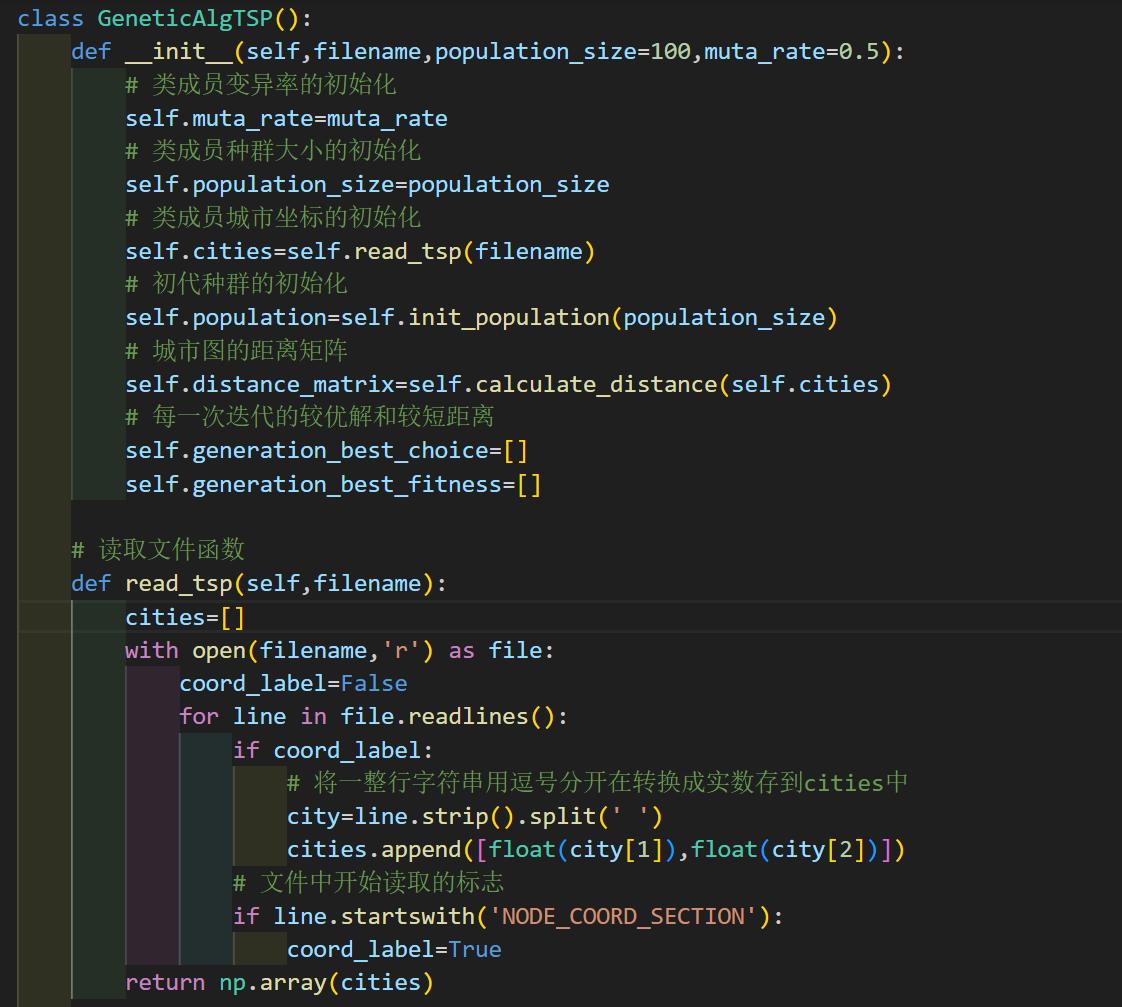
实现代码如下：



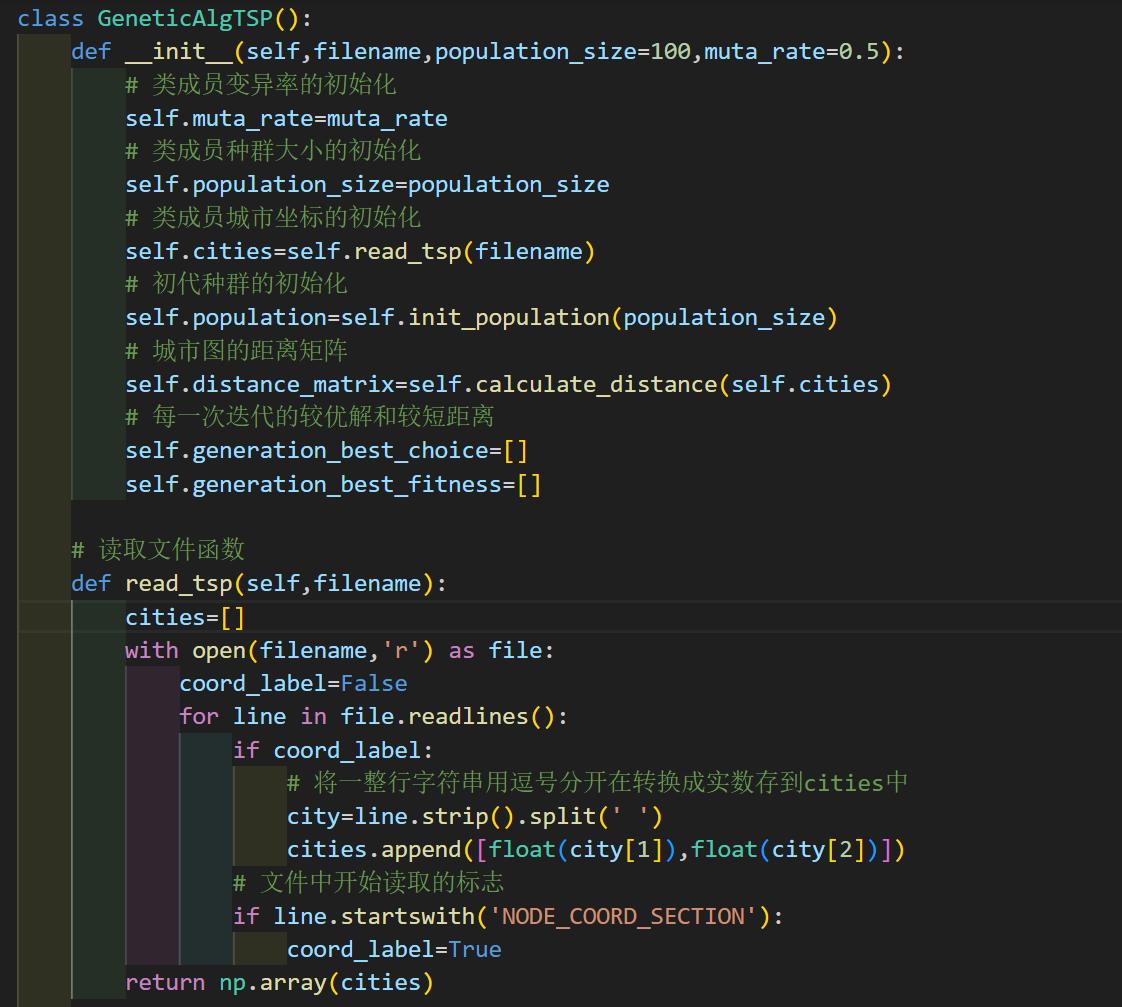


**④辅助函数**，由于我们是通过实现一个GeneticAlgTSP类来实现我们的遗传算法，所以需要一系列的函数实现初始化、计算适应度、将结果可视化

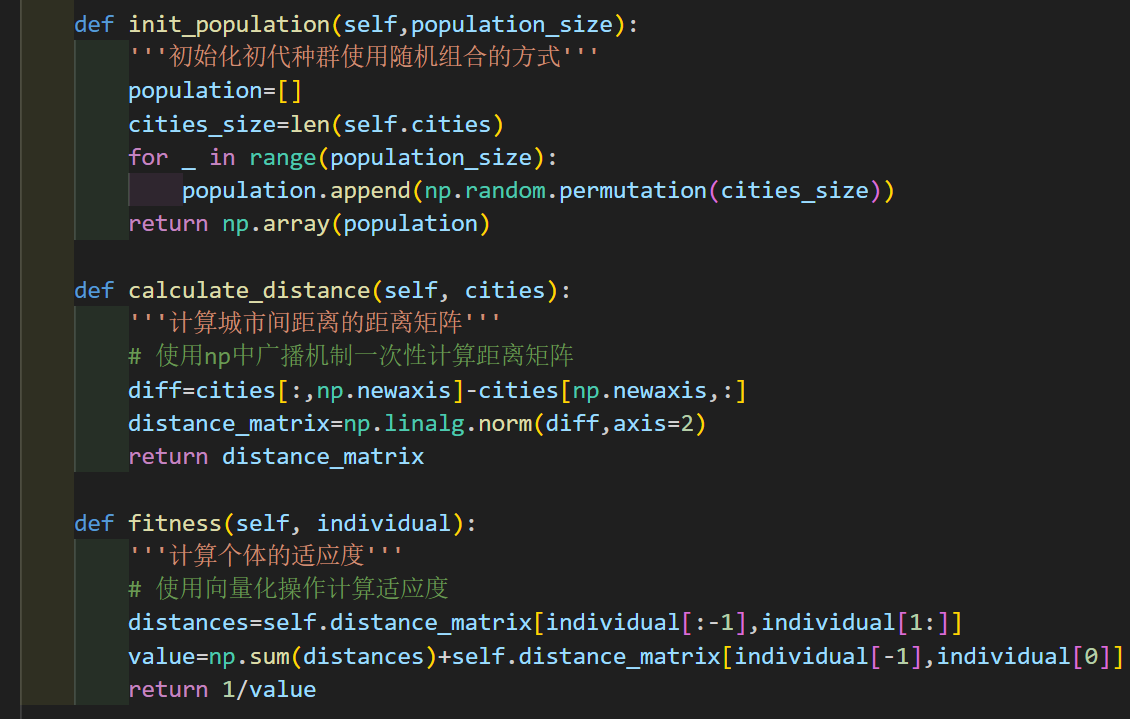
**初始化类函数：实现种群大小，城市坐标、距离矩阵、变异率的初始化**



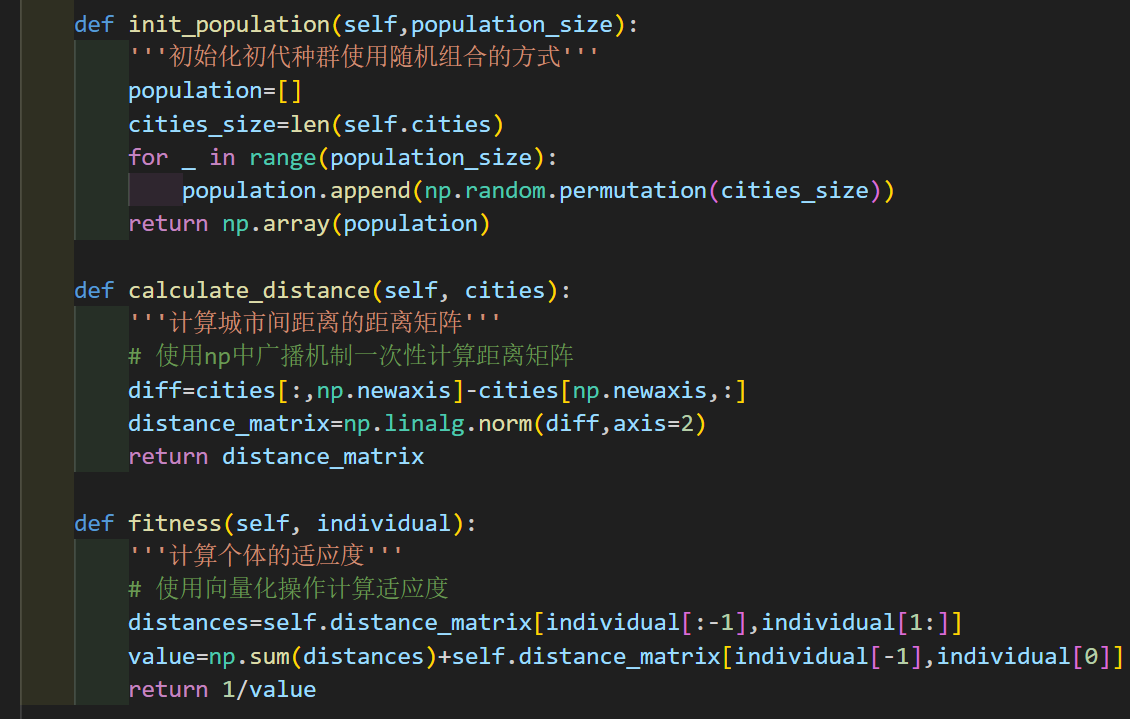
**read\_tsp函数：实现文件城市坐标的读取**



**init\_population函数**：**实现初代种群的编码**



**calculate\_distance函数和fitness函数**：采用np数据结构的特点，采用广播机制和向量化操作一次性的**计算所有城市间的距离，以及计算个体适应度**，提高算法迭代效率



1. 创新点&优化

启发式搜索：**使用了记忆性的线性冲突优化的曼哈顿距离**作为我们的启发值函数，该

启发式函数在曼哈顿距离的基础上对于线性冲突的情况，即**自身位置、目标位置都在同一行或者同一列**的的两个方块的分别在对方的移动路径上，这时**双方两个数字并不能直接穿过对方**导致普通的曼哈顿距离不够准确，为了解决这个线性冲突，我们给再给启发值增加2的惩罚项，也就是线性冲突的情况至少需要多移动2步才能够将双方移动到正确的位置，

比如[2,1,3,4]中的1和2发生的线性冲突，使用曼哈顿距离的话这里认为只需要移动2步实际上是不准确的，这个时候必须要从下面移动绕过对方才能够回到正确的位置，也就是最少需要移动4步才能够回到正确位置.

而**记忆性**，指的是可以发现我们每次转移状态，**只是和上一个状态变化了2个方块而已**，对于线性冲突的情况下也就是改变了两行或者两列的启发值情况，这里我想到了**动态规划dp的思路**，如果我们能够**记住上一个状态的启发值**，那么我们只需要计算这两行或者两列所**带来的值的变化**即可，**无需从头遍历节点计算启发值**，可以设想到随着规模越来越大25、36、...-puzzle，这种算法提高的效率就会越大.

# 实验结果及分析

1. 实验结果展示示例

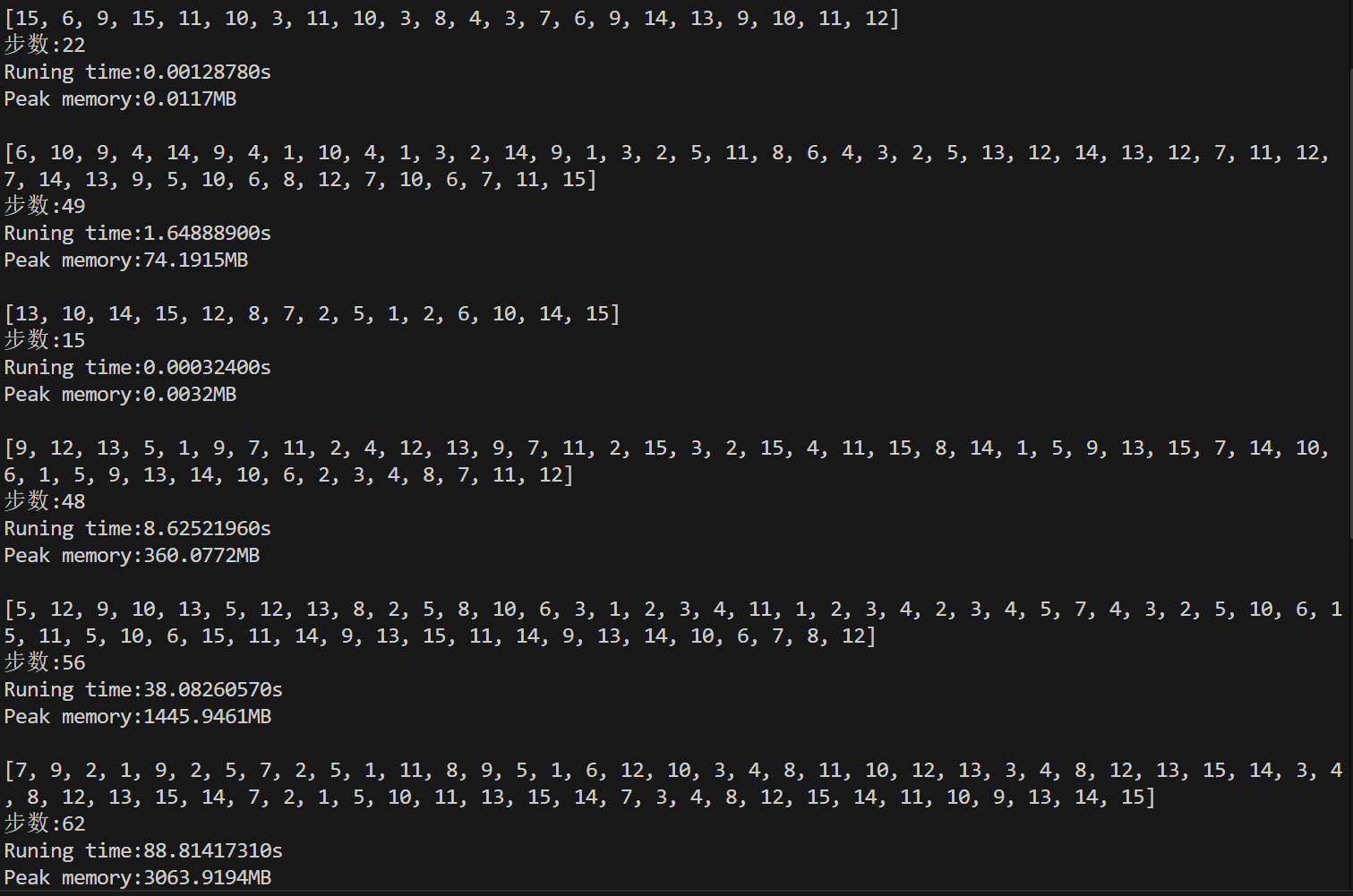
**启发式搜索解决15-Puzzle问题**

测试数据结果如下表：

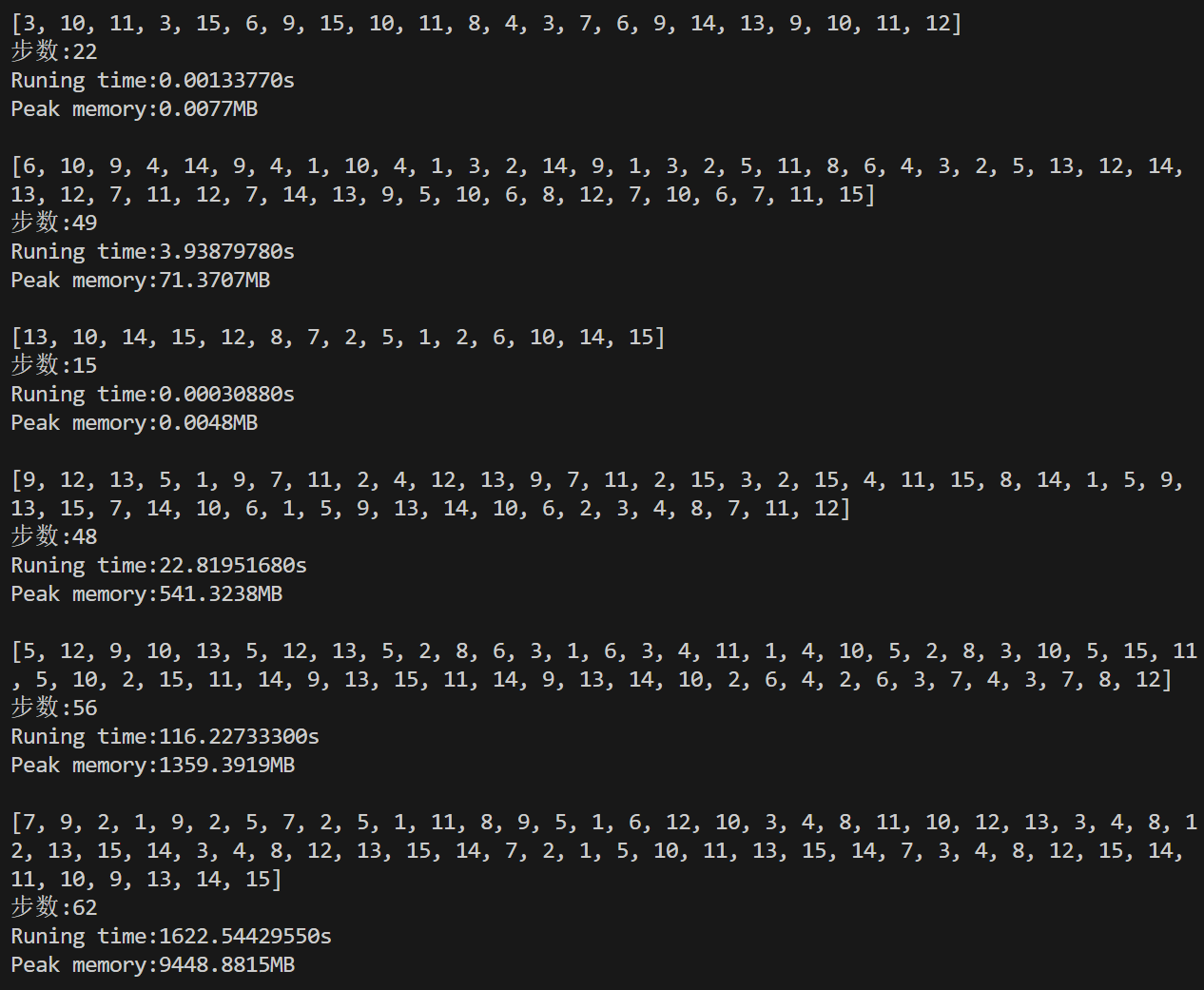
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | A\_star | | | IDA\_star | | |
| 测试例子序号 | 运行时间/s | 占用内存/MB | 步数 | 运行时间/s | 占用内存/MB | 步数 |
| 1 | 0.00128 | 0.0117 | 22 | 0.00133 | 0.0077 | 22 |
| 2 | 1.648 | 74.19 | 49 | 3.938 | 71.37 | 49 |
| 3 | 0.00032 | 0.0032 | 15 | 0.0003 | 0.0048 | 15 |
| 4 | 8.6252 | 360.077 | 48 | 22.82 | 541.32 | 48 |
| 5 | 38.082 | 1445.94 | 56 | 116.227 | 1359.39 | 56 |
| 6 | 88.814 | 3063.92 | 62 | 1622.544 | 9448.88 | 62 |

下面为程序的输出：

1. 算法



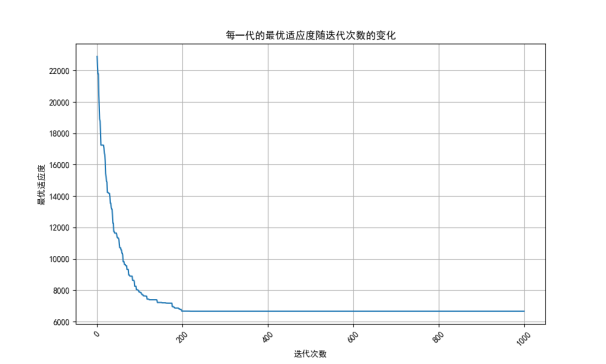
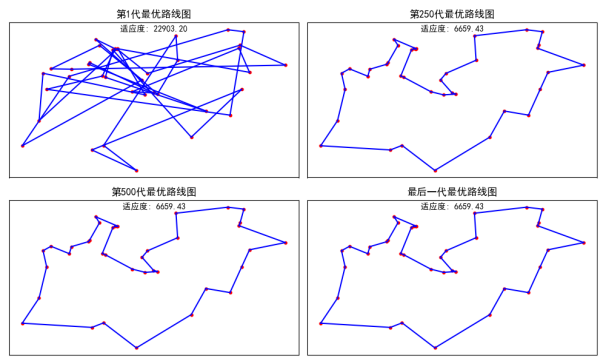
(2)算法



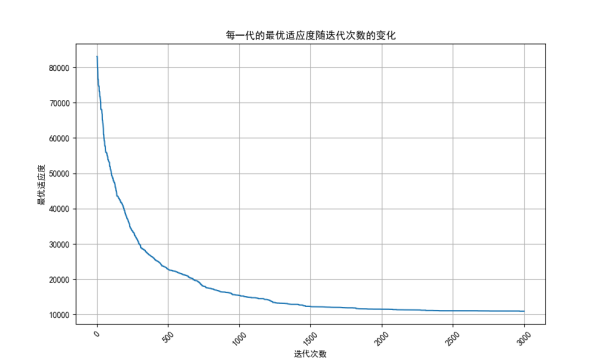
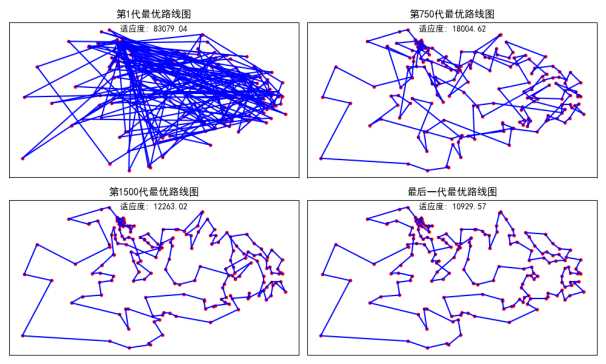
**遗传算法解决TSP问题**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 数据集 | 种群数 | 变异率 | 运行时间 | 迭代次数 | 所得的最短路程 | 实际最短路程 |
| dj38 | 100 | 0.5 | 34s | 1000 | 6659 | 6656 |
| qa197 | 100 | 0.5 | 3min22s | 3000 | 10929 | 9352 |
| uy734 | 100 | 0.5 | 34min45s | 10000 | 127778 | 79114 |

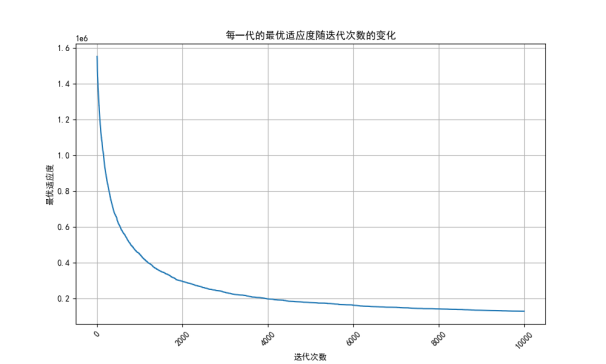
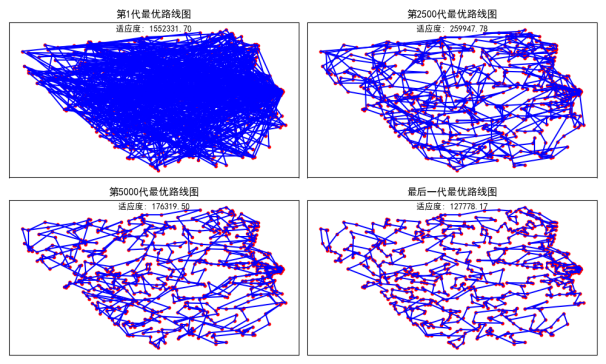
①dj38：



②qa197



③uy734



2. 评测指标展示及分析

**启发式搜索解决15-Puzzle问题**

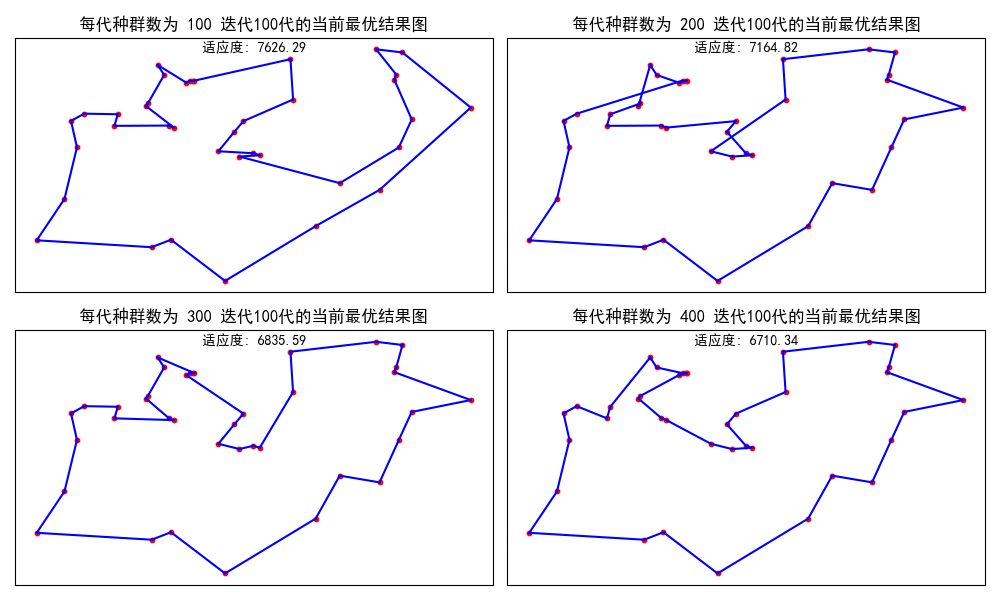
**不同启发式函数的比较：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 测试的例子为第二个测试用例即[[11,3,1,7],[4,6,8,2],[15,9,10,13],[14,12,5,0]] | | | | | |
|  | A\_star | | IDA\_star | |  |
|  | 运行时间/s | 占用内存/MB | 运行时间/s | 占用内存/MB |  |
| 错位方格数 | ∞ | 无 | ∞ | 无 |  |
| 曼哈顿距离 | 113 | 2398 | 117 | 2103 |  |
| 记忆曼哈顿距离 | 74 | 1877 | 128 | 1547 |  |
| 曼哈顿距离+线性冲突 | 38 | 1445 | 116 | 1359 |  |

**遗传算法解决TSP问题**

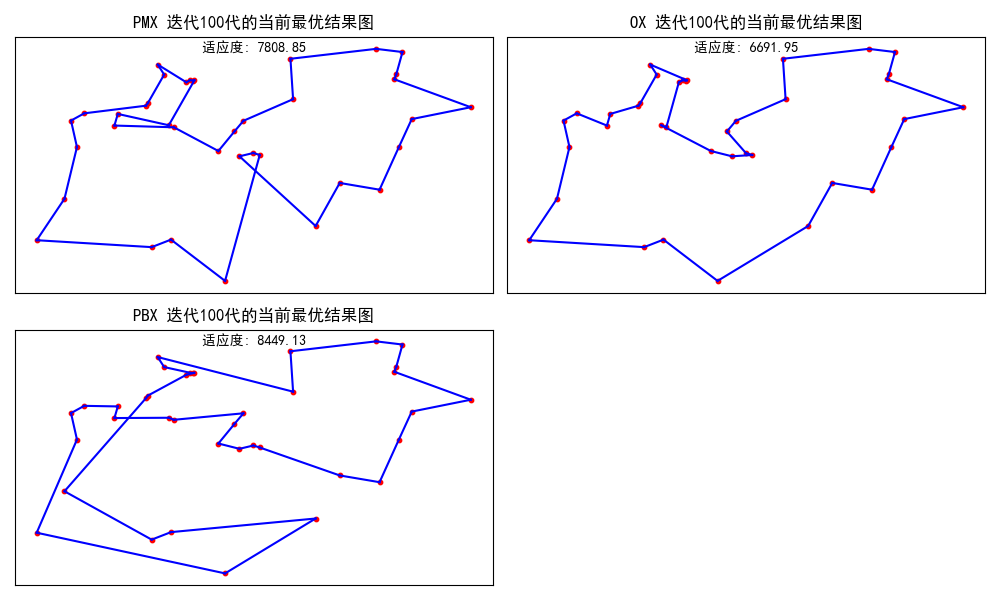
下面的测试数据如果无特殊说明均使用dj38，种群数量100，变异率0.5，PMX交叉函数

**不同种群数量的比较：**

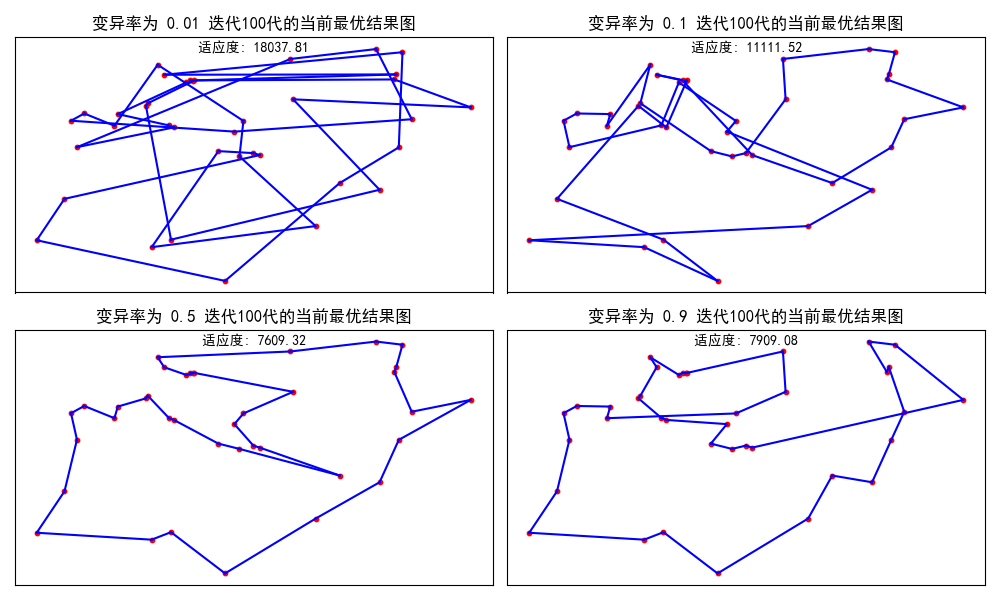


可以看到随着种群数量的增加，相同情况下，得出的结果更加的准确，但是种群数量的增加会带来效率问题，时间的增长是线性的，所以种群的数量不宜过大，也不能太小，太小的话会导致种群并没有发生太多的繁衍.

**不同交叉函数的比较：**



**不同变异率的比较：**



可以看到变异率太小的话会导致我们迭代的时候跳不出局部最优解，导致性能不好，但是当变异率过高的时候，可能会导致优良形状保存不下来，性能也会下降，所以应该保持一个较为适中的变异率.