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	

一、 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.

二、 实验内容

1. 算法原理

(1) 启发式搜索

评价函数的定义如下：

$$f(n) = h(n) + g(n)$$

其中$g(n)$表示从初始节点到达n节点的路劲成本，$h(n)$从n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$h(n)$**可采纳性意味着最优性**，在设计$h(n)$时往往需要考虑

- A^*算法：在深度优先搜索(BFS)的基础上，每次扩展探索评价函数$f(n)$最大的节点
- IDA^*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$f(n)$

(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

2. 伪代码

(1)A^*算法

Algorithm 1: A_star

```
Input:  s_0
Return: search_path
Initialize: frontier ← [s_0]    // 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)IDA^*算法

Algorithm 2: IDA_star

```
Input:  s_0
Return: search_path
Initialize: bound ← Heurisic(s_0) + g(s_0) // 初始 bound 值
while True do
    initialize: close ← None
    // 调用 dfs 获得最小花费
    min_cost ← dfs(s_0)
    // 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
```

3. 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

```python
def misplaced(puzzle):
    '''启发式函数1: 不在正确位置的方块个数'''
    value=0
    for index,num in enumerate(puzzle):
        if num == 0:
            continue
        target_x,target_y=target_coord[num]
        current_x,current_y=oneD2twoD(index)
        if (target_x,target_y) != (current_x,current_y):
            value+=1
    return value
```

②启发值函数 2：曼哈顿距离

```python
def manhattan(puzzle):
    '''启发式函数2: 曼哈顿距离'''
    value=0
    for index,num in enumerate(puzzle):
        if num == 0:
            continue
        target_x,target_y=target_coord[num]
        current_x,current_y=oneD2twoD(index)
        value+=abs(target_x-current_x)+abs(target_y-current_y)
    return value
```

③启发值函数 3：记忆性曼哈顿距离

```python
def optimized_manhattan(next_puzzle,move_block,pre_value):
    '''启发式函数3: 记忆性曼哈顿距离'''
    cur_value=0
    pre_index=next_puzzle.index(0)
    cur_index=next_puzzle.index(move_block)
    # 只需要计算移动的两个位置的曼哈顿距离的变化值即可
    target_x,target_y=target_coord[move_block]
    pre_x,pre_y=oneD2twoD(pre_index)
    cur_x,cur_y=oneD2twoD(cur_index)
    pre_h=abs(pre_x-target_x)+abs(pre_y-target_y)
    cur_h=abs(cur_x-target_x)+abs(cur_y-target_y)
    # 借用上一个状态的h值加上交换两个数字的h值得变化量得到新的h值
    cur_value=pre_value-pre_h+cur_h
    return cur_value
```

④启发值函数 4：线性冲突优化的曼哈顿距离

```python
def linear_conflict_manhattan(puzzle):
    '''启发式函数4：线性冲突优化的曼哈顿距离'''
    value=0
    for index,num in enumerate(puzzle):
        if num == 0:
            continue
        target_x,target_y=target_coord[num]
        current_x,current_y=oneD2twoD(index)
        '''
        对于同一行或者同一列的两个数字，如果他们分别出现在了对方的移动路径中，
        发生线性冲突，也就是至少需要多移动两次才能复原，故给h值加上惩罚项2
        '''
        if target_x == current_x:
            for k in range(current_y+1,4):
                other_x,other_y=target_coord[puzzle[current_x*4+k]]
                if other_y < target_y and other_x == current_x :
                    value+=2
        if target_y == current_y:
            for k in range(current_x+1,4):
                other_x,other_y=target_coord[puzzle[k*4+current_y]]
                if other_x < target_x and other_y == current_y :
                    value+=2
        value+=abs(target_x-current_x)+abs(target_y-current_y)
    return value
```

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：

```python
def get_next_states(puzzle):
    '''获得当前状态的邻接状态'''
    next_states=[]
    current_zero_index=puzzle.index(0)
    current_x,current_y=oneD2twoD(current_zero_index)
    # x,y坐标的变化量
    for dx,dy in [(0,1),(0,-1),(-1,0),(1,0)]:
        # 新坐标等于原坐标加上变化量
        next_x,next_y=current_x+dx,current_y+dy
        # 如果新坐标合法
        if 0 <= next_x < 4 and 0 <= next_y <4:
            new_puzzle=list(puzzle)
            next_zero_index=next_x*4+next_y
            # 记录移动的数字
            move_block=puzzle[next_zero_index]
            # 交换新坐标和原坐标的数字
            swap(new_puzzle[current_zero_index],new_puzzle[next_zero_index])
            next_states.append((tuple(new_puzzle),move_block))
    return next_states
```

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：

```python
def generate_path(puzzle,parents):
    '''实现从目标回溯路径的函数'''
    path=[]
    while parents[puzzle] != (None,None):
        pre_puzzle,move_block=parents[puzzle]
        path.append(move_block)
        puzzle=pre_puzzle
    return path[::-1]
```

(1) A^*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$f(n)=h(n)+g(n)$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$g(n)$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：

```python
def A_star(puzzle,target):
    # 初始化初始状态的数据结构、g、h值
    init_g=0
    # 启发式函数，这里选用线性冲突优化的曼哈顿距离
    init_h=linear_conflict_manhattan(puzzle)
    # frontier表
    frontier=[]
    # close表
    visited={puzzle:0}
    # 父亲节点列表
    parents=dict()
    parents[puzzle]=(None,None)
    # 初始化堆，元素为(f,puzzle,g,h)
    heapq.heappush(frontier,(init_g+init_h,puzzle,init_g,init_h))
    # A_star算法核心循环
    while frontier:
        # 当前节点
        _,cur_puzzle,g,h=heapq.heappop(frontier)
        # 找到目标返回路径
        if cur_puzzle == target:
            return generate_path(target,parents)
        # 遍历所有邻接节点
        for next_puzzle,move_block in get_next_states(cur_puzzle):
            # 如果不在close表，或者出现在close但是g值更优，加入frontier表
            if next_puzzle not in visited or visited[next_puzzle]>g+1:
                parents[next_puzzle]=(cur_puzzle,move_block)
                visited[next_puzzle]=g+1
                #next_h=optimized_manhattan(next_puzzle,move_block,h)
                next_h=linear_conflict_manhattan(next_puzzle)
                heapq.heappush(frontier,(g+1+next_h,next_puzzle,g+1,next_h))
    return []
```

(2) IDA^*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$f(n)$值作为本次迭代的上限(bound)值进行 dfs，代码如下：

```python
def dfs(puzzle,target,g,visited,parents,bound):
    # 计算当前状态的f值，这里选用线性冲突优化的曼哈顿距离
    f=linear_conflict_manhattan(puzzle)+g
    # 如果f值超过了当前的界限，返回f值
    if f>bound:
        return f
    # 如果当前状态等于目标状态，返回-1表示找到目标
    if puzzle == target:
        return -1
    min_cost=float('inf')
    # 遍历当前状态的所有可能的下一个状态
    for next_puzzle,move_block in get_next_states(puzzle):
        # 如果不在close表，或者出现在close但是g值更优，加入frontier表
        if next_puzzle not in visited or visited[next_puzzle] > g+1:
            visited[next_puzzle]=g+1
            parents[next_puzzle]=(puzzle,move_block)
            # 递归调用dfs，优先扩展出现的节点
            cost=dfs(next_puzzle,target,g+1,visited,parents,bound)
            if cost == -1:
                return -1
            # 如果当前的代价小于最小代价，更新最小代价
            if cost < min_cost:
                min_cost=cost
    return min_cost
```

```python
# IDA_star核心函数
def IDA_star(puzzle,target):
    # 初始上限
    bound=linear_conflict_manhattan(target)
    # IDA_star核心循环
    while 1:
        # 每次循环从初始点开始搜索，初始化close表
        visited={target:0}
        # 记录父亲节点
        parents=dict()
        parents[target]=(None,None)
        # 调用dfs深度搜索
        min_cost=dfs(target,target,0,visited,parents,bound)
        # 找到目标，返回搜索路径
        if min_cost == -1:
            return generate_path(target,parents)
        # 若未找到更新上限bound迭代加深上限f值
        bound=min_cost
    return []
```

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：

```python
def iterate(self,num_iterations):
    self.num_interations=num_iterations
    for i in trange(0,num_iterations):#trange
        childs=[]
        for _ in range(self.population_size//2):
            # 从当前种群中选两个亲本个体
            selection=self.selection()
            # 亲本个体杂交产生新的个体
            child1,child2=self.crossover(selection)
            # 产生子代可能发生变异
            child1,child2=self.mutation(child1),self.mutation(child2)
            # 将新产生的个体加入到childs
            childs.extend([child1,child2])
        # 选择新一代的种群
        new_population=np.concatenate((np.array(childs),self.population))
        # 根据适应度最大的前100个构成下一个种群(优胜劣汰)
        fitness_values = np.array([self.fitness(individual) for individual in new_population])
        top_indices = np.argpartition(-fitness_values,self.population_size)[:self.population_size]
        new_population = new_population[top_indices]
        # 记录当代的最优解和适应度
        best_individual = new_population[np.argmax(fitness_values[top_indices])]
        self.generation_best_choice.append(best_individual)
        self.generation_best_fitness.append(1/self.fitness(best_individual))
        #更新新一代的种群
        self.population=new_population
    return self.generation_best_choice[-1]
```

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：

```
# 选择个体的方式
def tournament_selection(self):
    '''随机竞争锦标赛选择法'''
    # 锦标赛规模，即每次从种群中选取的个体数量，可以根据需要调整
    tournament_size = 9
    fitnesses = np.array([self.fitness(individual) for individual in self.population])
    selected_indices = []
    for _ in range(2):
        # 从种群中随机选取个体组成锦标赛
        tournament_indices = np.random.choice(len(fitnesses),tournament_size)
        # 获取锦标赛中个体的适应度
        tournament_fitnesses = fitnesses[tournament_indices]
        # 找到锦标赛中适应度最高的个体的索引
        winner_index = tournament_indices[np.argmax(tournament_fitnesses)]
        selected_indices.append(winner_index)
    return self.population[selected_indices[0]], self.population[selected_indices[1]]
def roulette_selection(self):
    '''轮盘赌'''
    # 依据适应度占总适应度的比例为概率选择个体
    fitnesses=np.array([self.fitness(individual) for individual in self.population])
    total_fitness=sum(fitnesses)
    index1,index2=np.random.choice(len(fitnesses),size=2,p=fitnesses/total_fitness)
    return self.population[index1],self.population[index2]
```

②crossover 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：

```
# 交叉互换的方式
def PMX_cross(self,parents):
    parent1,parent2=parents
    child1,child2=np.copy(parent1),np.copy(parent2)
    # 随机选择两个位置作为交叉区域的起始和结束位置
    start,end=sorted(np.random.randint(0,len(child1),2))
    # 进行部分映射交叉操作，交换两个子代在交叉区域内的基因片段
    child1[start:end+1],child2[start:end+1]=parent1[start:end+1],parent2[start:end+1]
    # 解决交换后的冲突问题，采用映射的方式，使得个体合理化
    for index in range(len(child1)):
        if index < start or index > end:
            while child1[index] in child1[start:end+1]:
                match_index=np.where(child1[start:end+1] == child1[index])[0]
                if match_index.size > 0:
                    child1[index]=child2[match_index[0]+start]
            while child2[index] in child2[start:end+1]:
                match_index=np.where(child2[start:end+1] == child2[index])[0]
                if match_index.size > 0:
                    child2[index]=child1[match_index[0]+start]
    return child1,child2
def OX_Cross(self,parents):…
def PBX_Cross(self,parents):…
```

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：

```
# 变异的方式
def inversion_mutation(self,individual):
    '''倒置变异'''
    # 选择一个子序列将其倒置
    point1,point2=sorted(np.random.randint(0,len(individual),2))
    offspring=np.copy(individual)
    offspring[point1:point2+1]=offspring[point1:point2+1][::-1]
    return offspring
def insertion_mutation(self,individual):
    '''插入变异'''
    # 选一个数字移动到另一个位置
    src,pos=np.random.randint(0,len(individual),2)
    tem=np.concatenate((individual[:src],individual[src+1:]))
    offspring=np.concatenate((tem[:pos],[individual[src]],tem[pos:]))
    return offspring
def displacement_mutation(self,individual):
    '''位移变异'''
    # 选择一段移动到另一个位置
    point1,point2,pos=sorted(np.random.randint(0,len(individual),3))
    tem=np.concatenate((individual[:point1],individual[point2+1:]))
    offspring=np.concatenate((tem[:pos],individual[point1:point2+1],tem[pos:]))
    return offspring
```

```
def swap_mutation(self,individual):
    '''交换变异'''
    # 选择两个基因交换位置
    point1,point2=np.random.randint(0,len(individual),2)
    offspring=np.copy(individual)
    offspring[point1],offspring[point2]=offspring[point2],offspring[point1]
    return offspring
```

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

```
    def __init__(self,filename,population_size=100,muta_rate=0.5):
        # 类成员变异率的初始化
        self.muta_rate=muta_rate
        # 类成员种群大小的初始化
        self.population_size=population_size
        # 类成员城市坐标的初始化
        self.cities=self.read_tsp(filename)
        # 初代种群的初始化
        self.population=self.init_population(population_size)
        # 城市图的距离矩阵
        self.distance_matrix=self.calculate_distance(self.cities)
        # 每一次迭代的较优解和较短距离
        self.generation_best_choice=[]
        self.generation_best_fitness=[]
```

read_tsp 函数：实现文件城市坐标的读取

```
    # 读取文件函数
    def read_tsp(self,filename):
        cities=[]
        with open(filename,'r') as file:
            coord_label=False
            for line in file.readlines():
                if coord_label:
                    # 将一整行字符串用逗号分开在转换成实数存到cities中
                    city=line.strip().split(' ')
                    cities.append([float(city[1]),float(city[2])])
                # 文件中开始读取的标志
                if line.startswith('NODE_COORD_SECTION'):
                    coord_label=True
        return np.array(cities)
```

init_population 函数：实现初代种群的编码

```
    def init_population(self,population_size):
        '''初始化初代种群使用随机组合的方式'''
        population=[]
        cities_size=len(self.cities)
        for _ in range(population_size):
            population.append(np.random.permutation(cities_size))
        return np.array(population)
```

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

```
def calculate_distance(self, cities):
    '''计算城市间距离的距离矩阵'''
    # 使用np中广播机制一次性计算距离矩阵
    diff=cities[:,np.newaxis]-cities[np.newaxis,:]
    distance_matrix=np.linalg.norm(diff,axis=2)
    return distance_matrix

def fitness(self, individual):
    '''计算个体的适应度'''
    # 使用向量化操作计算适应度
    distances=self.distance_matrix[individual[:-1],individual[1:]]
    value=np.sum(distances)+self.distance_matrix[individual[-1],individual[0]]
    return 1/value
```

4. 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.

三、 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) A^*算法

```
[15, 6, 9, 15, 11, 10, 3, 11, 10, 3, 8, 4, 3, 7, 6, 9, 14, 13, 9, 10, 11, 12]
步数:22
Runing time:0.00128780s
Peak memory:0.0117MB

[6, 10, 9, 4, 14, 9, 4, 1, 10, 4, 1, 3, 2, 14, 9, 1, 3, 2, 5, 11, 8, 6, 4, 3, 2, 5, 13, 12, 14, 13, 12, 7, 11, 12, 7, 14, 13, 9, 5, 10, 6, 8, 12, 7, 10, 6, 7, 11, 15]
步数:49
Runing time:1.64888900s
Peak memory:74.1915MB

[13, 10, 14, 15, 12, 8, 7, 2, 5, 1, 2, 6, 10, 14, 15]
步数:15
Runing time:0.00032400s
Peak memory:0.0032MB

[9, 12, 13, 5, 1, 9, 7, 11, 2, 4, 12, 13, 9, 7, 11, 2, 15, 3, 2, 15, 4, 11, 15, 8, 14, 1, 5, 9, 13, 15, 7, 14, 10, 6, 1, 5, 9, 13, 14, 10, 6, 2, 3, 4, 8, 7, 11, 12]
步数:48
Runing time:8.62521960s
Peak memory:360.0772MB

[5, 12, 9, 10, 13, 5, 12, 13, 8, 2, 5, 8, 10, 6, 3, 1, 2, 3, 4, 11, 1, 2, 3, 4, 2, 3, 4, 5, 7, 4, 3, 2, 5, 10, 6, 15, 11, 5, 10, 6, 15, 11, 14, 9, 13, 15, 11, 14, 9, 13, 14, 10, 6, 7, 8, 12]
步数:56
Runing time:38.08260570s
Peak memory:1445.9461MB

[7, 9, 2, 1, 9, 2, 5, 7, 2, 5, 1, 11, 8, 9, 5, 1, 6, 12, 10, 3, 4, 8, 11, 10, 12, 13, 3, 4, 8, 12, 13, 15, 14, 3, 4, 8, 12, 13, 15, 14, 7, 2, 1, 5, 10, 11, 13, 15, 14, 7, 3, 4, 8, 12, 15, 14, 11, 10, 9, 13, 14, 15]
步数:62
Runing time:88.81417310s
Peak memory:3063.9194MB
```

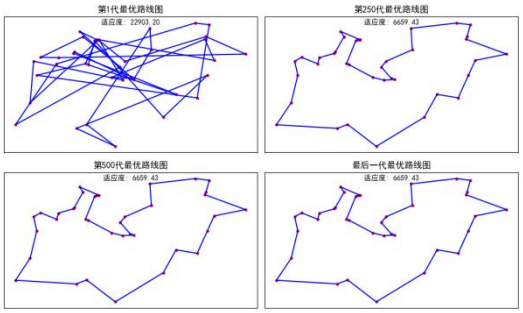
(2)IDA^*算法

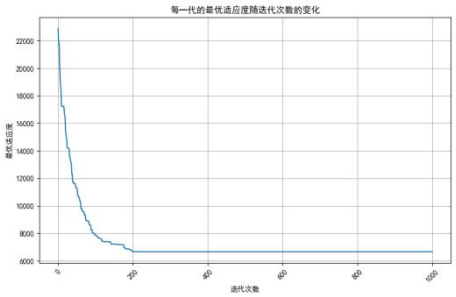
```
[3, 10, 11, 3, 15, 6, 9, 15, 10, 11, 8, 4, 3, 7, 6, 9, 14, 13, 9, 10, 11, 12]
步数:22
Runing time:0.00133770s
Peak memory:0.0077MB

[6, 10, 9, 4, 14, 9, 4, 1, 10, 4, 1, 3, 2, 14, 9, 1, 3, 2, 5, 11, 8, 6, 4, 3, 2, 5, 13, 12, 14, 13, 12, 7, 11, 12, 7, 14, 13, 9, 5, 10, 6, 8, 12, 7, 10, 6, 7, 11, 15]
步数:49
Runing time:3.93879780s
Peak memory:71.3707MB

[13, 10, 14, 15, 12, 8, 7, 2, 5, 1, 2, 6, 10, 14, 15]
步数:15
Runing time:0.00030880s
Peak memory:0.0048MB

[9, 12, 13, 5, 1, 9, 7, 11, 2, 4, 12, 13, 9, 7, 11, 2, 15, 3, 2, 15, 4, 11, 15, 8, 14, 1, 5, 9, 13, 15, 7, 14, 10, 6, 1, 5, 9, 13, 14, 10, 6, 2, 3, 4, 8, 7, 11, 12]
步数:48
Runing time:22.81951680s
Peak memory:541.3238MB

[5, 12, 9, 10, 13, 5, 12, 13, 5, 2, 8, 6, 3, 1, 6, 3, 4, 11, 1, 4, 10, 5, 2, 8, 3, 10, 5, 15, 11, 5, 10, 2, 15, 11, 14, 9, 13, 15, 11, 14, 9, 13, 14, 10, 2, 6, 4, 2, 6, 3, 7, 4, 3, 7, 8, 12]
步数:56
Runing time:116.22733300s
Peak memory:1359.3919MB

[7, 9, 2, 1, 9, 2, 5, 7, 2, 5, 1, 11, 8, 9, 5, 1, 6, 12, 10, 3, 4, 8, 11, 10, 12, 13, 3, 4, 8, 12, 13, 15, 14, 3, 4, 8, 12, 13, 15, 14, 7, 2, 1, 5, 10, 11, 13, 15, 14, 7, 3, 4, 8, 12, 15, 14, 11, 10, 9, 13, 14, 15]
步数:62
Runing time:1622.54429550s
Peak memory:9448.8815MB
```

遗传算法解决 TSP 问题

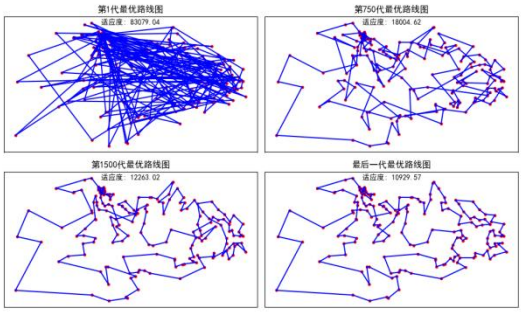
数据集	种群数	变异率	运行时间	迭代次数	所得的最短路程	实际最短路程
dj38	100	0.5	34s	1000	6659	6656
qa197	100	0.5	3min22s	3000	10929	9352
uy734	100	0.5	34min45s	10000	127778	79114

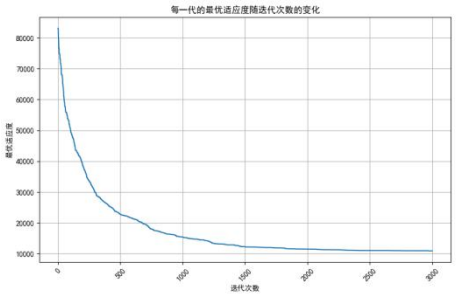
①dj38：



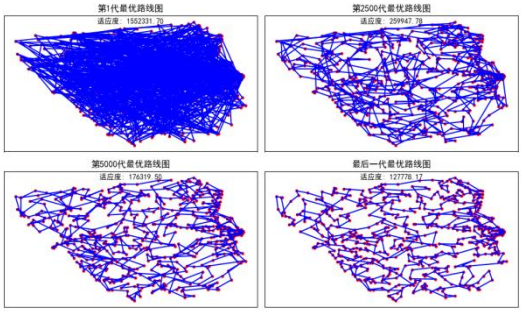


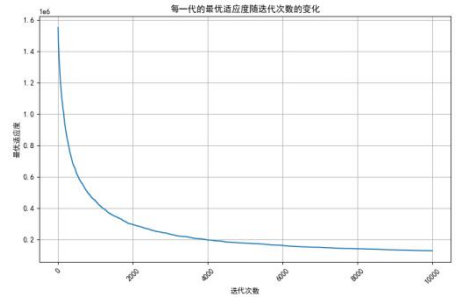
②qa197





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：

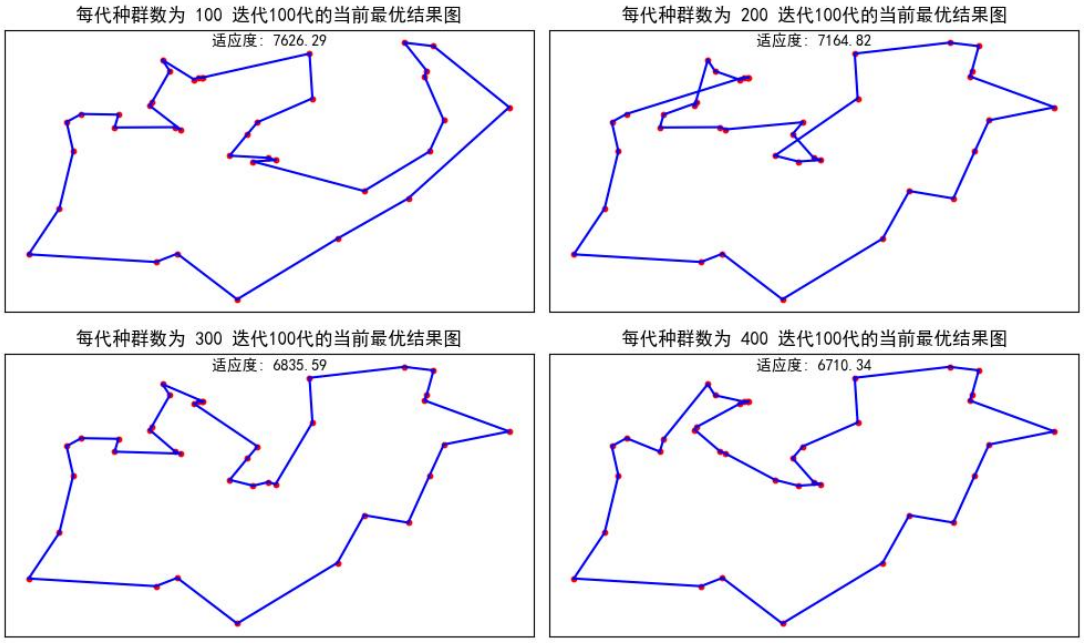
测试的例子为第二个测试用例即[[11,3,1,7],[4,6,8,2],[15,9,10,13],[14,12,5,0]]

	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

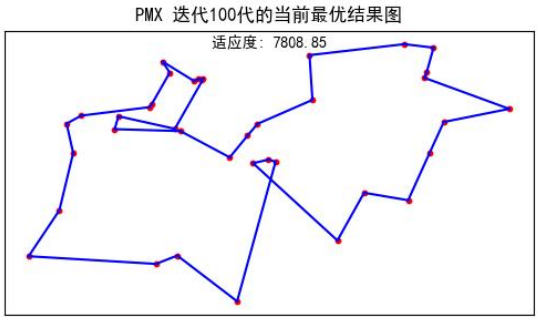
下面的测试数据如果无特殊说明均使用 `dj38`，种群数量 `100`，变异率 `0.5`，PMX 交叉函数

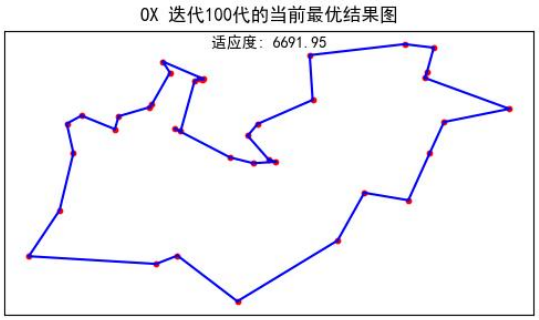
不同种群数量的比较：

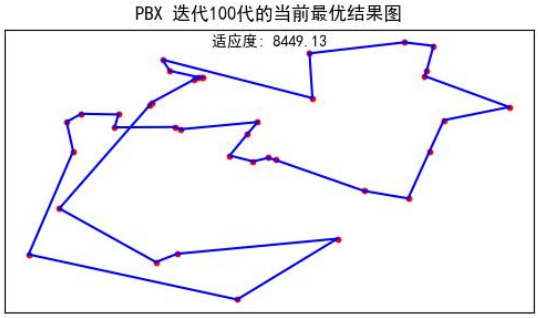


可以看到随着种群数量的增加，相同情况下，得出的结果更加的准确，但是种群数量的增加会带来效率问题，时间的增长是线性的，所以种群的数量不宜过大，也不能太小，太小的话会导致种群并没有发生太多的繁衍.

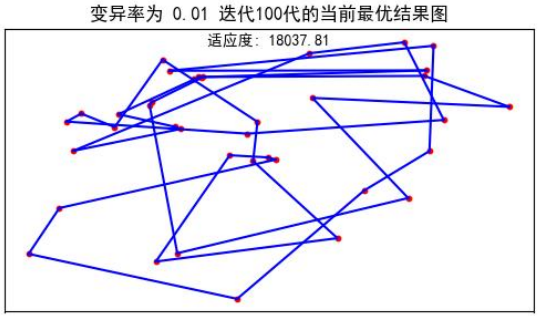
不同交叉函数的比较：

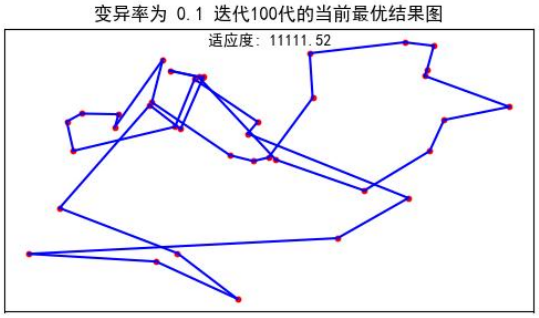


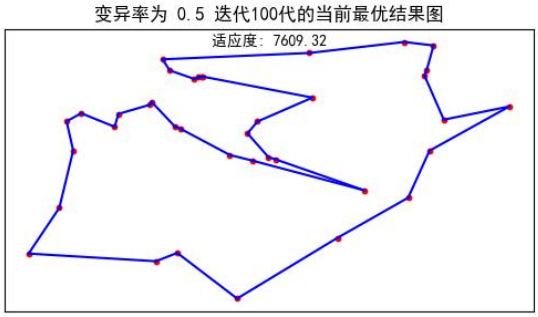


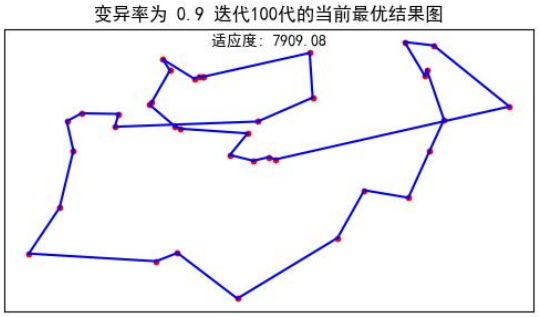


不同变异率的比较：









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.