中山大学计算机学院 本科生实验报告（2023学年春季学期）

课程名称：Artificial Intelligence 人工智能

教学班级		专业（方向）	
学号 2233 6173		姓名 罗弘杰	

实验题目 week5&6 A* & IDA*

实验任务·项目

- 使用A*与IDA*算法求15-Puzzle问题的**最优解**，启发式函数可以自己选取，可以多尝试几种不同的启发式函数，完成**压缩包的4个测例**及**ppt上的4个测例**（最后两个难度较高）
- 结果分析要求
  - 对比分析A*和IDA*的性能和结果（Memory Out请附上内存限制）
  - 如果使用了多种启发式函数也可以进行对比和分析
- 加分项（供参考）
  - 算法实现优化分析（使用数据结构、利用性质的剪枝等）
  - 未提及的启发式函数实现、对比和分析
- 启发式函数参考
  - 曼哈顿距离 $|x_1 - x_2| + |y_1 - y_2|$
  - 切比雪夫距离 $\max(|x_1 - x_2|, |y_1 - y_2|)$
- Deadline
  - 两周，4月8日 [周一] 23:59
- 提交命名
  - E3_学号.zip

实验内容

算法原理：

A*算法：

定义起点和终点，以及一个评价函数

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

其中，g(n)是从起点到结点n的路径的代价，通常设置为1/单位距离；

h(n)又叫做启发式函数，是从n到终点的一个启发式估计，这个函数存在一个隐式的完美启发式函数，如果h(n)越接近这个完美启发式，则评价函数效果越好，找到最优解的速度越快；

A*算法设置open和close，每次从open中获取f(n)最小的的结点，拓展其，**并将其拓展出的结点不在close表中的加入到open表**，循环此操作，直到找到终点或者open表为空。

h(n)的条件：

可采纳性： 总是低估从当前结点到终点的真实代价，且代价总大于0，不能任意小。 **可采纳性蕴含最优性**（若最优解存在，则能找到最优解）

一致性/单调性： 若对于任意节点n1,n2,总有

$$h(n1) <= c(n1 -> n2) + h(n2)$$

可以证明：**一致性蕴含可采纳性**

IDA*算法：

定义起点和终点，依然使用启发式函数和代价函数；

IDA*每次设置一个代价函数阈值，根据这个阈值，深度优先搜索结点的每一个子节点，若每一个子节点的代价函数都大于阈值，则将阈值更新为子节点中最小的代价函数，迭代搜索，直到遇到目标节点。

当h是可采纳的，可以证明IDA*是能保证找到最优解的

伪代码

```
A*(起点，  终点):
    将起点放入open表
    当open不为空：
        从open表中获取f(n)最小的结点current，将current放入close
        若current就是终点：
            返回
        否则：
            拓展current的子节点，
            将子节点不在close表中的放入open表中
            将子节点放入close表
    若open表为空：
        返回无路径
```

```
IDA_search(node, g, bound)
    f = g + h(node)
    若(f>bound):
        return f
    若(node == goal):
        return FOUND
    min = infinity;
    for each child in child_generator(node):
        t = IDA_search(child, g+1, bound)  //
        if(t == FOUNd):
            return FOUND
        if(t < min):
            min = t
    return min

IDA*(start, goal):
    将start放入open栈（stack）
    bound = h(start)
    while(true):
        t = IDA_search(start, 0, bound)
        if(t == FOUND):
            return bound
        if (t==infinity):
            return FALSE
        bound = t
```

关键代码展示（带注释）

结构优化前的A_Star和IDA_Star算法

```
'''
A* algorithm, before optimalized
'''
class node:
    def __init__(self, state : np.array, blank: tuple, g_cost: int, h_cost : int,
parent = None):
        self.blank = blank    #空白块的位置
        self.state = state    #当前状态,矩阵
        self.g_cost = g_cost  #从初始状态到当前状态的代价
        self.h_cost = h_cost  #从当前状态到目标状态的代价
        self.parent = parent

    def f_cost(self):
        return self.g_cost + self.h_cost

#获取四个方向的移动，返回一个列表包含四个方向的坐标元组
def try_moving(state, blank: tuple):
    blank_i, blank_j = blank[0], blank[1]
    moves = []
    if blank_i > 0:
        moves.append((blank_i - 1, blank_j))
    if blank_i < 3:
        moves.append((blank_i + 1, blank_j))
    if blank_j > 0:
        moves.append((blank_i, blank_j - 1))
    if blank_j < 3:
        moves.append((blank_i, blank_j + 1))
    return moves

#交换两个位置的数值,返回移动后的state 和 blank的位置
def swap(state, blank, move: tuple):
    new_state = copy.deepcopy(state)
    new_state[blank[0]][blank[1]], new_state[move[0]][move[1]] =
new_state[move[0]][move[1]], new_state[blank[0]][blank[1]]
    return new_state, move

# A*算法
def A_Star(state : np.array):
    # find the blank in the state
    for i in range(4):
        for j in range(4):
            if state[i][j] == 0:
                blank = (i, j)
    open_list = []
    closed_list = set()
    start_node = node(state, blank, 0, H(state)) #建立初始状态结点
    # 以下使用堆排序来实现优先队列，排序标准是代价函数和结点的id
    heapq.heappush(open_list, (start_node.f_cost, id(start_node), start_node))  #
将初始状态结点放入open_list
    while open_list:
```

```
        current_node = heapq.heappop(open_list)[2]  #取出open_list中代价最小的结点,2
表示元组中的第三个元素
        if np.array_equal(current_node.state, goal):
            print("done!")
            current_blank = current_node.blank
            current_node = current_node.parent
            movement = []
            while current_node:
                movement.append(current_node.state[current_blank])  #记录移动
                current_blank = current_node.blank
                current_node = current_node.parent
            movement.reverse()
            return movement  #返回移动
        #set会使用hash函数来判断同元素的state是否存在,作为环检测
        closed_list.add(tuple(map(tuple, current_node.state)))
        # 以下是对当前结点的四个方向进行搜索
        for move in try_moving(current_node.state, current_node.blank):
            new_state, new_blank= swap(current_node.state, current_node.blank,
move)
            if tuple(map(tuple, new_state)) not in closed_list:
                new_g_cost = current_node.g_cost + 1 #上一个状态代价函数的加一步
                new_h_cost = H(new_state) #新状态到目标状态的代价
                new_node = node(new_state, new_blank, new_g_cost, new_h_cost,
current_node)
                heapq.heappush(open_list, (new_node.f_cost(), id(new_node),
new_node))    #将新结点放入open_list
    return None
```

使用了np.array数据结构来存储数码矩阵，open表使用列表，**close表使用了set结构（由于python集合内置的哈希函数性能比较好，能大幅度加快close表的环检测）**,对于open表，我使用了heapq的堆排序来进行筛选最优结点；

结构优化后的A_star和IDA_star算法：查阅网上博客以后，有观点认为**使用tuple比使用list来存储数码矩阵有更多性能优势；**

--"存储数码状态的数据结构均为List，Tuple相对与list来说，在存储空间上显得更加轻量级一些。对于元组这样的静态变量，如果它不被使用并且占用空间不大时，Python 会暂时缓存这部分内存。这样，下次我们再创建同样大小的元组时，Python 就可以不用再向操作系统发出请求，去寻找内存，而是可以直接分配之前缓存的内存空间，这样就能大大加快程序的运行速度。"

除了元组，我还使用了generator，生成器的方法来优化程序。

使用生成器（generator）来动态生成子状态，而不是一次性生成所有可能的子状态。这种动态生成子状态的方法可以显著降低内存消耗，并且能够在需要时立即生成下一个子状态，从而提高了算法的效率。

from 【人工智能大作业】A*和IDA*搜索算法解决十五数码（15-puzzle）问题 （Python实现）（启发式搜索） astar算法15数码问题-CSDN博客

以下我参考使用了其方法重构了两个算法的实现，并进行了性能比较

```
'''
    A* algorithm
    implement the A* algorithm by using tuple, because the tuple is hashable and
compressible in python.
```

```
'''
def Hx(state : np.array):   #定义了曼哈顿启发式函数
    distance = 0
    for i in range(16):
            if state[i] != 0:
                x_state, y_state = i // 4, i % 4
                x_goal, y_goal = (state[i]- 1) // 4, (state[i]- 1) % 4
                distance += abs(x_goal - x_state) + abs(y_goal - y_state)
    return distance

OPEN = []
CLOSE = set()
path = []
way = []

def print_path(node, move = 15):  #打印移动路径
    if node.parent != None:  #深度优先搜索
        print_path(node.parent, node.state.index(0))
    path.append(node.state)
    way.append(node.state[move])
    return path, way

def child_generator():  #孩子结点生成器
    movetable = []
    for i in range(16):
        x,y = i%4, i//4
        moves = []
        if x > 0: moves.append(-1)
        if x < 3: moves.append(+1)
        if y > 0: moves.append(-4)
        if y < 3: moves.append(+4)
        movetable.append(moves)
    def children(state):
        idxz = state.index(0)
        l = list(state)
        for m in movetable[idxz]:
            l[idxz] = l[idxz + m]
            l[idxz + m] = 0
            yield (1, tuple(l))
            l[idxz + m] = l[idxz]
            l[idxz] = 0
    return children

node_sum = 0    #扩展结点数量
goal = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0)
class node:
    def __init__(self, state ,  g_cost: int, h_cost : int, parent = None):
        self.state = state    #当前状态,矩阵
        self.g_cost = g_cost  #从初始状态到当前状态的代价
        self.h_cost = h_cost  #从当前状态到目标状态的代价
        self.f_cost = self.g_cost + self.h_cost
        self.parent = parent

    def __lt__(self, other):
        if self.f_cost == other.f_cost:
            return self.g_cost > other.g_cost
```

```
        return self.f_cost < other.f_cost

def A_Star(start : tuple, Hx):  #A*算法主体
    OPEN = []
    CLOSE = set()
    root = node(start,  0, Hx(start), None)
    heapq.heappush(OPEN, root)
    while OPEN:
        top = heapq.heappop(OPEN)  #获取最佳结点
        global node_sum
        node_sum += 1
        if top.state == goal:
            return print_path(top)

        CLOSE.add(top.state)  #扩展的结点加入到close
        generator = child_generator()
        for cost, state in generator(top.state):
            if  state in CLOSE:  #环检测
                continue
            child = node(state, top.g_cost + cost, Hx(state), top)
            heapq.heappush(OPEN, child)
```

```
'''
IDA_star using tuple and generator
'''
OPEN = []
CLOSE = set()
path = []
way = []

def child_generator():   #孩子结点生成器
    movetable = []
    for i in range(16):
        x,y = i%4, i//4
        moves = []
        if x > 0: moves.append(-1)
        if x < 3: moves.append(+1)
        if y > 0: moves.append(-4)
        if y < 3: moves.append(+4)
        movetable.append(moves)
    def children(state):
        idxz = state.index(0)
        l = list(state)
        for m in movetable[idxz]:
            l[idxz] = l[idxz + m]
            l[idxz + m] = 0
            yield (1, tuple(l))
            l[idxz + m] = l[idxz]
            l[idxz] = 0
    return children

node_sum = 0
```

```
goal = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0)
class node:
    def __init__(self, state ,  g_cost: int, h_cost : int, parent = None):
        self.state = state    #当前状态,矩阵
        self.g_cost = g_cost  #从初始状态到当前状态的代价
        self.h_cost = h_cost  #从当前状态到目标状态的代价
        self.f_cost = self.g_cost + self.h_cost
        self.parent = parent

    def __lt__(self, other):
        if self.f_cost == other.f_cost:
            return self.g_cost > other.g_cost
        return self.f_cost < other.f_cost

def IDAsearch(g, Hx, bound):
    global node_sum
    node = path[-1]  # 取出path中的start结点,这里相当于如果迭代超过了bound，回来继续迭代的开始结点，也就是path中的最后一个结点
    node_sum +=1
    f = g + Hx(node)
    if f > bound:       # 如果f(n)的值大于bound则返回f(n)
        return f
    if node == goal: # 目标检测，以0表示找到目标
        return 0

    Min = 99999 # 保存子结点中返回的最小的f值，作为下次迭代的bound
    generator = child_generator() # 获取child_generator()中生成的child
    for cost, state in generator(node):
        if state in CLOSE: continue
        path.append(state)
        way.append(state.index(0))
        CLOSE.add(state) # 利用set查找的优势，进行路径检测
        t = IDAsearch(g+1,Hx,bound)

        if t == 0: return 0
        if t < Min: Min = t

        path.pop() # 回溯
        way.pop()
        CLOSE.remove(state)
    return Min


def IDAstar(start, Hx):
    global CLOSE
    global path
    bound = Hx(start) # IDA*迭代限制
    path = [start] # 路径集合，视为栈
    CLOSE = {start}


    while(True):
        ans = IDAsearch(0, Hx,bound) # path，g，Hx，bound，调用搜索函数，获得孩子结点的启发式最小阈值
        if(ans == 0):
            return (path,way,bound)
```

```
        if ans == -1:
            return None
        bound = ans # 此处对bound进行更新
```

创新点&优化

1. 代码实现上的优化：使用了可以想到的方法尽可能优化程序的性能，包括**close表使用集合的数据机构，open表使用堆排序来排序结点，使用tuple来存储数码矩阵来优化内存和加快哈希计算，使用了generator方法**来优化每次生成结点的速度
2. 启发式函数上的创新：我尝试了多种启发式函数，曼哈顿距离，静态分区可加数据库（static disjoint addictive pattern database）等启发式函数

   曼哈顿距离：通过求和每一个非0滑块到目标的横轴距离和纵轴距离来计算启发式函数，该函数已经在课堂上被证明是单调的，因此是可采纳的，可以找到最优解。然而使用这个函数距离完美启发式函数距离比较远，因为他没有考虑滑块之间相互的阻碍，也就是线性冲突。对于极端的情况，使用该函数无法获得最优解，要么是时间太久，要么是内存占的过高。

   模式数据库：我从15 Puzzle (michael.kim)的个人博客上认识到这种方法，并查阅了其发明者的论文：

   ```
   AdditivePatternDatabaseHeuristics,&amp;quot;JournalofAIResearch,Vol.22,pp.279-318,2004.
   ```

   该方法通过分区预计算数码问题的最短路径（作者使用从目标状态BFS回溯到每一个情况的方法），并将之存储到数据库中，命名为模式数据库，具体的分类可以有7-8, 6-6-3, 5-5-5, 4-4-4-3分法。对于越少的分区，计算的情况越多，计算复杂性越高。举个例子：对于一个7-8分区中的8模块，你需要计算16到9的乘积个情况，共518，918，400个单元，注意对于BFS,到后面的寻找困难是越来越高的。而对于4-4-4-3分区中的4模块，你需要计算16到13的乘积个情况，总共43，680个单元，这显然是更友好的。对于我的个人电脑，**计算一个5模块大致需要3个小时，一个4模块大致需要1个小时。**由于计算困难，其他分区情况我还没有尝试，除了4-4-4-3和5-5-5分区。更少的分区对于完美启发式函数的逼近效果是更好的（如果只有一个分区，实际上是将所有的情况都用数据库存储下俩，然而这在实现上是难以想象的：）

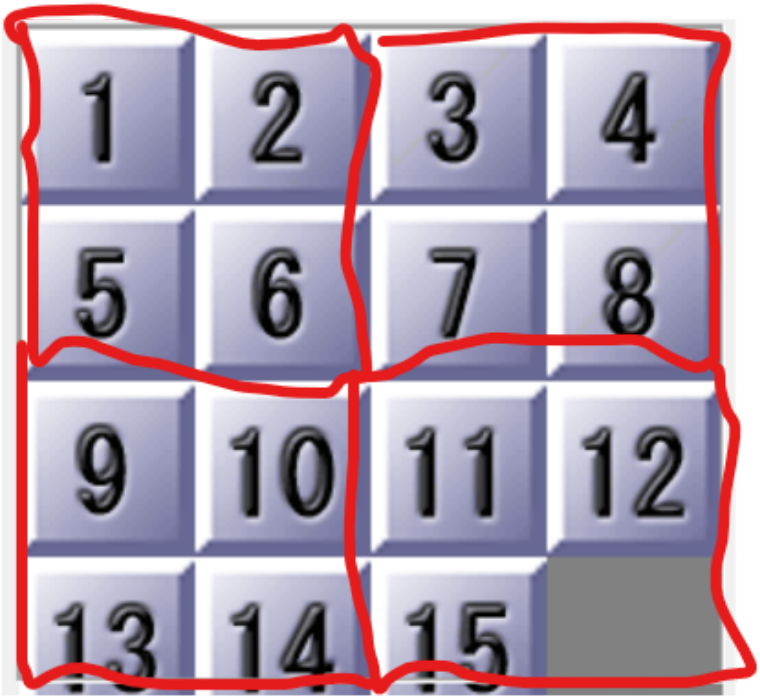
   具体解释：

   以下是我三分区的情况，看goalstate是分区内注意的结点，在计算启发式距离时，只计算这些滑块。其他滑块置为-1，**（这本质上是对现实情况的放松！）**，预计算每个分区所有情况的最短距离，并将其加入到数据库，以后就可以**不用计算，查询得到启发式距离**

```
subset_1_goalstate = {'values': [1, 2, 5, 6, 9],   #这是三分区的情况，最后会在distance中生成搜索字典，以加快启发式函数计算
                      'goalstate': [[1, 2, -1, -1],
                                    [5, 6, -1, -1],
                                    [9, -1, -1, -1],
                                    [-1, -1, -1, 0]],
                      'goalstate_indexes': ((0, 0), (0, 1), (1, 0), (1, 1), (2, 0)),
                      'distances': {((0, 0), (0, 1), (1, 0), (1, 1), (2, 0)): 0}
                      }

subset_2_goalstate = {'values': [3, 4, 7, 8, 12],
                      'goalstate': [[-1, -1, 3, 4],
                                    [-1, -1, 7, 8],
                                    [-1, -1, -1, 12],
                                    [-1, -1, -1, 0]],
                      'goalstate_indexes': ((0, 2), (0, 3), (1, 2), (1, 3), (2 ,2)),
                      'distances': {((0, 2), (0, 3), (1, 2), (1, 3), (2 ,3)): 0}
                      }

subset_3_goalstate = {'values': [10, 11, 13 ,14, 15],
                      'goalstate': [[-1, -1, -1, -1],
                                    [-1, -1, -1, -1],
                                    [-1, 10, 11, -1],
                                    [13, 14, 15, 0]],
                      'goalstate_indexes': ((2, 1), (2, 2), (3, 0),(3, 1),(3, 2)),
                      'distances': {((2, 1), (2, 2), (3, 0),(3, 1),(3, 2)): 0}
                      }
```



对于4分区，我将用**以上的分区方法**来进行实验，对于每一个分区，**最好应该使得分区内部之间的联系被充分考虑**，所以我将他们分为整齐的4个模块。

对于3分区，我将用以上的分法来实现。

模式数据库生成和使用代码（我将用3分区的来说明，4分区的类同）

```
#生成3分区数据库
n = 4
def get_subset_positions(state, values_to_find):
            '''
            这个函数输入状态，获取其中部分结点的位置
            Inputs: state (list of lists): the state of the puzzle
                    values_to_find (list): the values to find in the state. E.g.
[1, 2, 4, 5, 13]
            Outputs: positions (tuple): the positions of the values in the state.
E.g. ((0, 1), (0, 2), (1, 0), (1, 1), (3, 2))
            '''

            values_poistion_dict = {value: (0, 0) for value in values_to_find}

            for i in range(n):
                for j in range(n):
                    if state[i][j] in values_to_find:
                        values_poistion_dict[state[i][j]] = (i, j)

            return tuple(values_poistion_dict.values())

def static_additive_disjoint_pattern_database():
    '''
    这个函数通过从目标结点使用宽度优先搜索回溯每一个状态，记录BFS的最短路径，最后将其存储为字
典，放在pkl文件中
    '''
    try:  #尝试打开数据库
        with open('static_additive_disjoint_pattern_database.txt', 'r') as f:
```

```
            static_additive_disjoint_pattern_database = eval(f.read())
            print('Static additive disjoint pattern database loaded from file.')
    except:  #否则开始生成数据库
        print('Static additive disjoint pattern database not found. Generating
static additive disjoint pattern database...')
        file_name = ['pattern_database_3_1.pkl', 'pattern_database_3_2.pkl',
'pattern_database_3_3.pkl']  #生成文件的名字
        from copy import deepcopy

        n = 4

        subset_1_goalstate = {'values': [1, 2, 5, 6, 9],   #这是三分区的情况，最后会
在distance中生成搜索字典，以加快启发式函数计算
                              'goalstate': [[1, 2, -1, -1],
                                            [5, 6, -1, -1],
                                            [9, -1, -1, -1],
                                            [-1, -1, -1, 0]],
                              'goalstate_indexes': ((0, 0), (0, 1), (1, 0), (1,
1), (2, 0)),
                              'distances': {((0, 0), (0, 1), (1, 0), (1, 1), (2,
0)): 0}
                              }

        subset_2_goalstate = {'values': [3, 4, 7, 8, 12],
                              'goalstate': [[-1, -1, 3, 4],
                                            [-1, -1, 7, 8],
                                            [-1, -1, -1, 12],
                                            [-1, -1, -1, 0]],
                              'goalstate_indexes': ((0, 2), (0, 3), (1, 2), (1,
3), (2 ,2)),
                              'distances': {((0, 2), (0, 3), (1, 2), (1, 3), (2
,3)): 0}
                              }

        subset_3_goalstate = {'values': [10, 11, 13 ,14, 15],
                              'goalstate': [[-1, -1, -1, -1],
                                            [-1, -1, -1, -1],
                                            [-1, 10, 11, -1],
                                            [13, 14, 15, 0]],
                              'goalstate_indexes': ((2, 1), (2, 2), (3, 0),(3,
1),(3, 2)),
                              'distances': {((2, 1), (2, 2), (3, 0),(3, 1),(3,
2)): 0}
                              }

        moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]  #移动方向
        K= -1
        for subset in [subset_1_goalstate]:
            K += 1
            #以下是宽度优先搜索，计算每一个情况最早出现的步骤
            queue = [] # initialize the queue
            visited = set() # initialize the visited set

            queue.append(subset['goalstate'])

            visited.add(tuple([tuple(row) for row in subset['goalstate']]))
```

```
            while queue:
                state = queue.pop(0)
                indexes = get_subset_positions(state, subset['values'])
                distance = subset['distances'][indexes]

                for i in range(n):   #获取空白块位置
                    for j in range(n):
                        if state[i][j] == 0:
                            blank_row, blank_col = i, j
                            break

                for move in moves:
                    new_row, new_col = blank_row + move[0], blank_col + move[1]
                    if (new_row >= 0 and new_row < n) and (new_col >= 0 and
new_col < n):
                        new_state = deepcopy(state) #深复制原结点
                        new_state[blank_row][blank_col], new_state[new_row]
[new_col] = new_state[new_row][new_col], new_state[blank_row][blank_col]
                        indexes = get_subset_positions(
                            new_state, subset['values']) # 获取状态的key


                        if tuple([tuple(row) for row in new_state]) not in
visited:  #环检测

                            if new_state[new_row][new_col] == -1: #检查是否交换的不
是分区内的，即-1
                                if indexes in subset['distances']: # 若状态已经被加
入，更新最近的距离
                                    subset['distances'][indexes] = min(
                                        distance, subset['distances'][indexes])

                                else: #否则直接加入字典
                                    subset['distances'][indexes] = distance

                            else:  #如果交换的是分区内的

                                if indexes in subset['distances']:  # 若状态已经被
加入，更新最近的距离
                                    subset['distances'][indexes] = min(
                                        distance+1, subset['distances'][indexes])
                                else:
                                    subset['distances'][indexes] = distance + 1 #
移动了一步，距离加1

                            queue.append(new_state) # 将新结点加入队列
                            visited.add(tuple([tuple(row)
                                    for row in state])) #加入到visited
            print("Subset finished. ")

            import pickle  #保存到pkl文件
            with open(file_name[0], 'wb') as f:
                pickle.dump(
                    [subset], f)
                print(subset['values'], end = '')
```

```
                print('Static additive disjoint pattern database saved to file.')
                f.close()

        return subset_1_goalstate['distances'], subset_2_goalstate['distances'],
subset_3_goalstate['distances']



static_additive_disjoint_pattern_database()
```

```
#使用数据库
#coding=gbk
'''
这个文件用于计算3个子集的模式数据库，之前计算的距离存储在数据字典中。
你可以在pattern_database_3_1.pkl文件等中检查它们。
- 输入：状态为元组
- 输出：状态的启发式值
'''
import pickle
n = 4
# 读取.pkl文件
with open('pattern_database_3_1.pkl', 'rb') as f:
    data1 = pickle.load(f)

with open('pattern_database_3_2.pkl', 'rb') as f:
    data2 = pickle.load(f)

with open('pattern_database_3_3.pkl', 'rb') as f:
    data3 = pickle.load(f)


# 使用读取的数据

def get_subset_positions(state, values_to_find):
    """遍历状态并返回值列表中值的位置

    输入：
        - state（列表的列表）：谜题的状态
        - values_to_find（列表）：要在状态中查找的值。例如 [1, 2, 4, 5, 13]
    输出：
        - positions（元组）：状态中值的位置。例如 ((0, 1), (0, 2), (1, 0), (1, 1), (3,
2))
    """

    # 初始化字典
    values_position_dict = {value: (0, 0) for value in values_to_find}

    # 遍历状态并将值的位置添加到字典中
    for i in range(16):
        if state[i] in values_to_find:
            values_position_dict[state[i]] = (i//4, i%4)

    # 从字典中作为元组返回状态中值的位置
```

```
    return tuple(values_position_dict.values())

def pattern_db_3(state: tuple):
    '''
    这个函数计算了4个子集中瓦片的距离之和([1, 2, 5, 6, 9] [3, 4, 7, 8, 12] [10, 11, 13,
14, 15])
    先前计算的距离存储在数据字典中。

    输入:
    - state（列表的列表）：谜题的状态

    输出:
    - distance（整数）：子集中瓦片的距离之和
    '''
    # 获取子集中瓦片的位置
    indexes_1 = get_subset_positions(state, data1[0]['values'])
    indexes_2 = get_subset_positions(state, data2[0]['values'])
    indexes_3 = get_subset_positions(state, data3[0]['values'])
    # 从子集中获取距离
    distance_1 = data1[0]['distances'][indexes_1]
    distance_2 = data2[0]['distances'][indexes_2]
    distance_3 = data3[0]['distances'][indexes_3]
    # 返回距离之和
    return distance_1 + distance_2 + distance_3

# 测试为主函数
# 测试速度
if __name__ == "__main__":
    import time
    state = (6, 10, 3, 15, 14, 8, 7, 11, 1, 0, 9, 2, 5, 13, 12, 4)
    begin = time.perf_counter()
    print(pattern_db_3(state))
    end = time.perf_counter()
    print("time:", end-begin)
```

数据库生成可以使用一些优化方法，比如使用同构的思想，可以将一个3模块本来的16，15，14相乘种情况减少为16x15x14/3/2/1种，这是将内部的3种的排列进行置换以减少计算量，具体可以参考

https://github.com/mwong510ca/15Puzzle_OptimalSolver/blob/master/PatternElement.java%20-%20details.md

由于时间有限，我没有使用该方法，使用了朴素的BFS从目标结点回溯到每一种情况。

使用数据库计算启发式函数时，会根据每一个分区内的滑块来计算该情况在数据库中对于的最短距离，将每一个分区的距离相加就是对启发式函数的估计。

模式数据库的可采纳性证明：

条件：1，每个分区是不相交的；2，每次移动，只会影响一个分区的计算结果，因为每个分区都是不相交的，每次移动滑块只会改变一个分区的启发式函数计算。

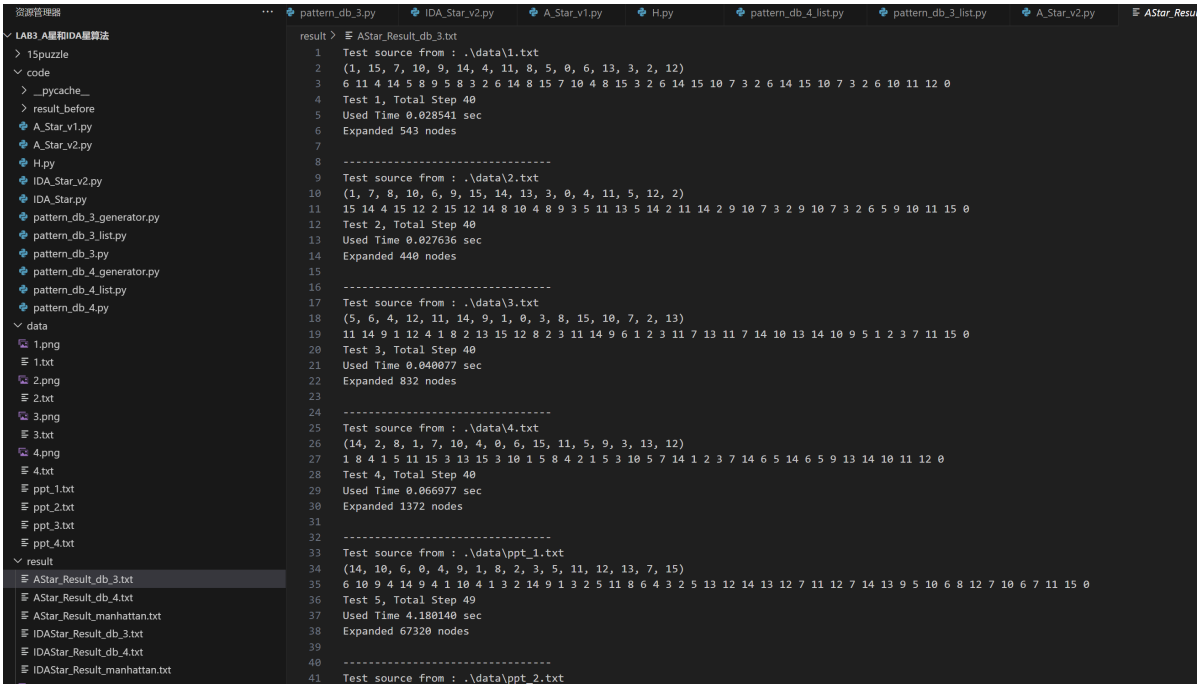
证明：从单调性证明，由单调性就可以说明可采纳性。怎么证明单调性？可以证明每次移动，启发式函数最多减少1次.这样的话，思路就很清晰：由于模式数据库的条件，每次移动只会改变一个分区的启发式计算，然后对于这个分区，只有一个滑块移动（也就是和空白块交换位置），启发式函数要么不变，要么减少1。反证法：若减少2，那么说明原先数据库的计算出现了错误，**（因为完全可以通过少2的情况移动一步获得上一步情况的最短路径）**

由此，模式数据库可以保证获得最优解。

实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

具体更多请查询文件夹result内的txt文件夹，总共有6个结果文件.



2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

拓展深度：标黄的表示不是最优解

算法和启发式	1	2	3	4	ppt1	ppt2	ppt3	ppt4
A* manhattan	40	40	40	40	49	48		
IDA* manhattan	40	40	40	40	49			
A* pattern db 3	40	40	40	40	49	48	56	62
IDA* pattern db 3	40	40	42	40	49	48	56	62
A* pattern db 4	40	40	40	40	49	50	56	62
IDA* pattern db 4	40	40	40	40	49	48	56	62

解释：模式数据库经过证明已经被验证是满足可采纳性的， 至于为什么出现无法找到最优解；可能的原因是生成数据库时出现小bug

扩展结点总数：

算法和启发式	1	2	3	4	ppt1	ppt2	ppt3	ppt4
A* manhattan	3727	1440	569	12014	438265	2569406		
IDA* manhattan	23512	18965	8081	256270	4825776			
A* pattern db 3	543	440	832	1372	67320	188278	947129	1452643
IDA* pattern db 3	5097	3258	2649	22023	806077	1768748	11743468	25720899
A* pattern db 4	671	761	69	1260	29382	212584	1271568	2019805
IDA* pattern db 4	5944	5027	615	19538	303173	1424562	21629186	66784776

运行时间/s：

算法和启发式	1	2	3	4	ppt1	ppt2	ppt3	ppt4
A* manhattan	0.2142	0.0479	0.0144	0.2799	13.0292	92.2905		
IDA* manhattan	0.3529	0.2658	0.1165	3.1645	62.3952			
A* pattern db 3	0.0285	0.0276	0.0401	0.0670	4.1801	11.5731	59.6147	90.5252
IDA* pattern db 3	0.0604	0.0392	0.0444	0.2752	9.5268	21.8015	131.0332	272.00
A* pattern db 4	0.0588	0.0499	0.0043	0.0675	1.4605	13.3381	71.2226	106.5786
IDA* pattern db 4	0.0838	0.0862	0.0086	0.3007	4.4351	19.9884	309.0956	1622.4972

元组，生成器优化前后的A*算法时间比较

算法和启发式	1	2	3	4	ppt1	ppt2
A* manhattan 优化前	1.4534	0.1995	0.2565	3.0808	176.8320	419.7346
A* manhattan 优化后	0.2142	0.0479	0.0144	0.2799	13.0292	92.2905

从结果可以看出，

1： **A*算法由于不需要重复遍历结点，速度比IDA_star快很多**， 扩展结点也会少很多；

2：使用曼哈顿启发式函数，由于扩展结点随探索深度的增加呈现指数增加，会导致无法支撑运算；

3：使用模式数据库能大大加速程序执行， 因为其作为启发式函数更接近完美启发式，能有效探索结点，从而减少需要扩展的结点数目，同时，注意到，**分区的减少，使得少分区的数据库比起更多分区的性能更高；**

4：使用**元组和生成器能大幅度加快A*算法**的实现

参考资料

PS：可以自己设计报告模板，但是内容必须包括上述的几个部分，不需要写实验感想

1 AdditivePatternDatabaseHeuristics,"JournalofAIResearch,Vol.22,pp.279-318,2004., // pattern database 思想源头

2 【人工智能大作业】A*和IDA*搜索算法解决十五数码（15-puzzle）问题 （Python实现）（启发式搜索） astar算法15数码问题-CSDN博客 //代码结构优化

3 https://github.com/mwong510ca/15Puzzle_OptimalSolver/blob/master/PatternElement.java%20-%20details.md // 模式数据库加快生成