中山大学计算机学院

人工智能

本科生实验报告

（2023 学年春季学期）

课程名称：Artificial Intelligence

教学班级	20230349	专业（方向）	计算机科学与技术(系统结构)
学号	22336290	姓名	张超

一、 实验题目

利用 A*搜索算法和 IDA*搜索算法解决 15-puzzle 问题

二、 实验内容

1. 算法原理

A*算法：A*算法是一种常用于图形搜索和路径规划的启发式搜索算法。它在寻找从起点到终点的最短路径时非常有效。A*算法结合了 Dijkstra 算法的广度优先搜索和贪婪最佳优先搜索的优点，通过启发式评估函数（heuristic evaluation function）来指导搜索方向，从而在保证找到最优路径的前提下提高了搜索效率。

以下是 A*算法的基本原理：

- **启发式函数（Heuristic Function）：**A*算法的核心是启发式函数，它用于评估节点的“优良程度”。这个函数通常记作(f(n) = g(n) + h(n))，其中：

  1. (g(n))表示从起始节点到节点(n)的实际路径代价（即已经花费的路径长度）。
  2. (h(n))表示从节点(n)到目标节点的估计代价，即启发式评估函数的估计值。

- **开放列表（Open List）：**A*算法通过维护一个开放列表来存储待探索的节点。每次迭代时，算法从开放列表中选择最优节点进行探索。
- **关闭列表（Closed List）：**已经被探索的节点会被移动到关闭列表中，以防止重复探索。
- **算法步骤：**

  1. 初始化：将起点加入开放列表，并将其(f)值设为初始启发式估计值。
  2. 迭代：重复以下步骤直到达到终点或开放列表为空：

1. 从开放列表中选择当前(f)值最小的节点。
2. 如果当前节点是终点，则算法终止，路径被找到。
3. 将当前节点从开放列表移至关闭列表。
4. 对当前节点的相邻节点进行评估：
    1. 如果相邻节点不在开放列表中，则将其加入，并计算其(f)值。
    2. 如果相邻节点已经在开放列表中，检查当前路径是否更优。如果是，则更新其(g)值和(f)值。

3. 路径重建：一旦达到终点，可以从终点开始沿着父节点指针（通常在节点对象中存储）反向追溯，直到回到起点，从而找到最优路径。

A*算法的关键在于选择一个合适的启发式函数 h(n)。良好的启发式函数应该能够提供尽可能接近实际最优路径长度的估计值，同时尽量减少搜索空间，以提高算法的效率。

IDA*算法：IDA*（Iterative Deepening A*）是 A 算法的一种变体，它通过迭代加深搜索的方式在保证找到最优路径的前提下，降低了内存消耗。与 A 算法一样，IDA*也是一种启发式搜索算法，用于解决图形搜索和路径规划问题。

以下是 IDA*算法的基本原理：

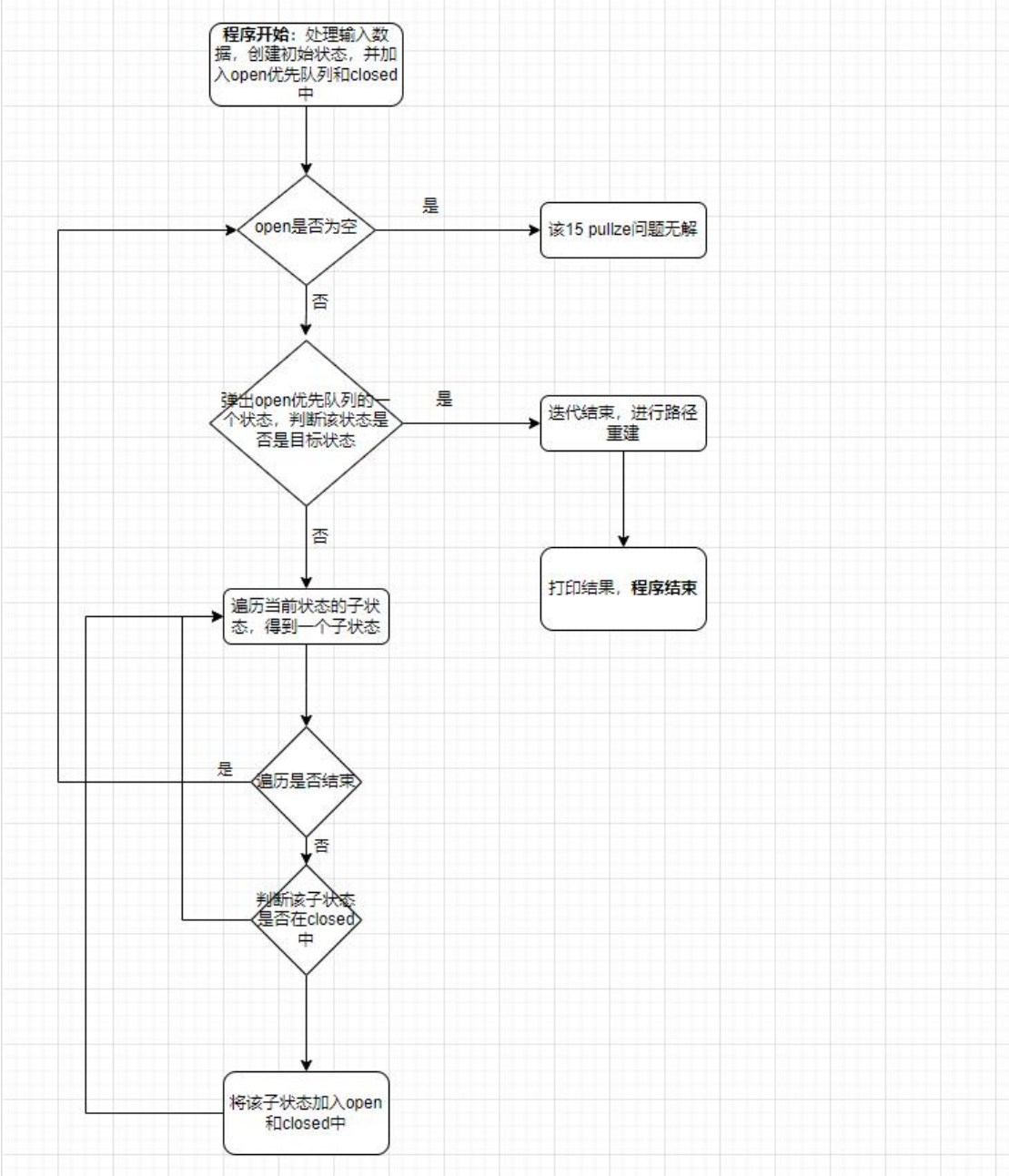
- **启发式函数（Heuristic Function）：**与 A 算法一样，IDA 算法也使用启发式函数来指导搜索。启发式函数评估节点的估计代价，并帮助算法决定搜索方向。
- **阈值限制（Limitation）：**IDA*算法采用阈值限制来控制搜索的深度。在每一轮迭代中，搜索的阈值限制逐渐增加，直到找到目标节点为止。
- **算法步骤：**

1. 初始化：将起点加入搜索树，设置初始阈值限制为 0。
2. 迭代搜索：重复以下步骤直到找到目标节点为止：
   (1) 使用深度优先搜索（DFS）在当前深度限制内搜索。
   (2) 如果找到目标节点，则算法终止，路径被找到。
   (3) 如果未找到目标节点，增加深度限制，并重新开始搜索。
   (4) 如果当前状态的估价函数值大于阈值，则记录当前状态的估价函数值（以确定阈值的下一个迭代值），回溯
3. 路径重建：一旦找到目标节点，可以从目标节点开始向上回溯，直到回到起点，从而找到最优路径。

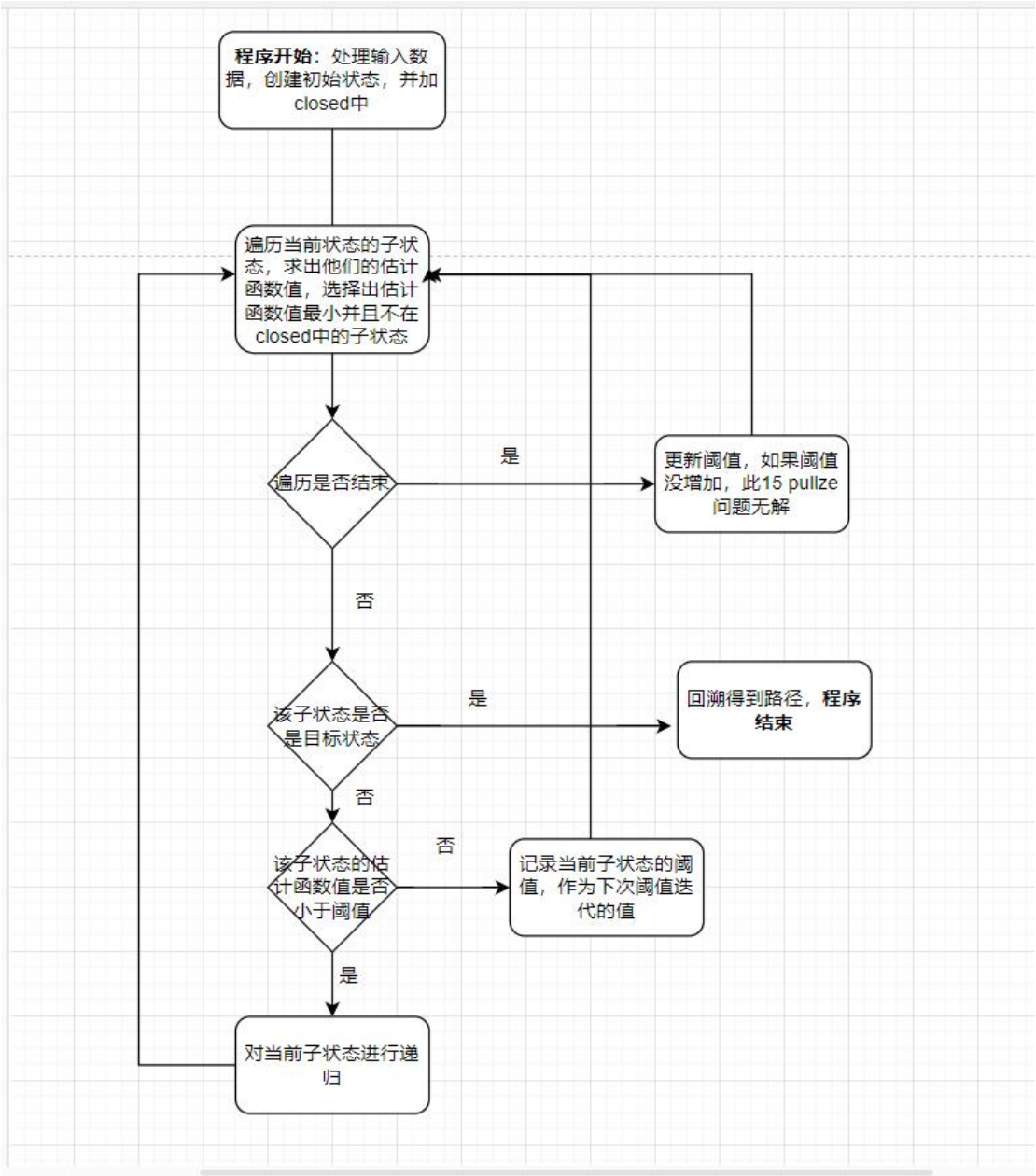
IDA 算法的主要优点在于它不需要维护显式的开放列表和关闭列表，从而降低了内存消耗。但与此同时，IDA 可能会多次重复搜索相同的路径，因为在每次迭代中都会重新搜索整个阈值范围（这里我做了改进，不需要重新搜索）。因此，IDA 算法在某些情况下可能比 A 算法更慢，尤其是在搜索空间较大且启发式函数不够准确时。

2. 伪代码

A*算法流程图：



IDA*算法流程图：



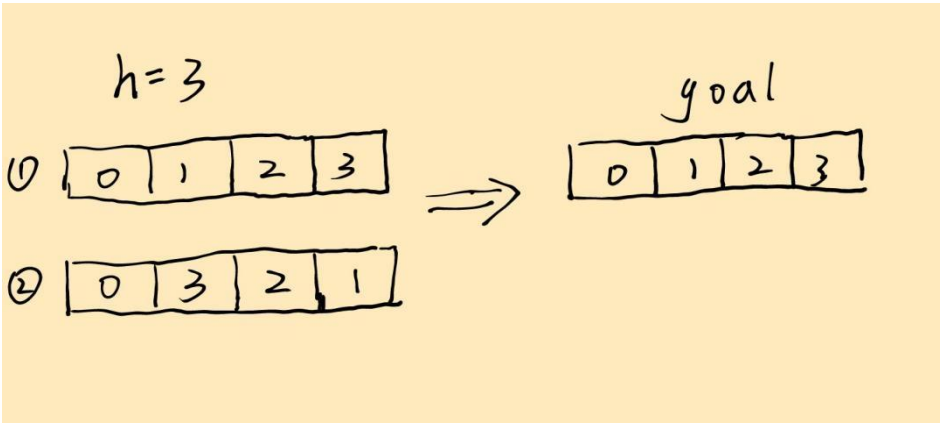
3. 关键代码展示（带注释）

下面所展示的代码为已经优化过的代码

- **启发式函数**

错位的方块数量:在这个启发式函数中，需要统计当前状态下被错置的数码块之和，其值作为当前状态的 H(n)。这种启发式函数很简单，但是由于其 h 的取值范围只有 0-15（算上 0 的话就是 16），取值范围太小，完全不足以描述出当前状态要达到目标状态的难易程度。

如图所示：



两个状态的启发值都是 3（不算 0），但是很显然，上面状态相较于下面状态来说，要更容易达到目标状态。因此选用错位格子个数作为启发式函数是效率很低的。这个启发式函数实现很简单而且很没用，所以之后的测试都不会使用该启发式函数。

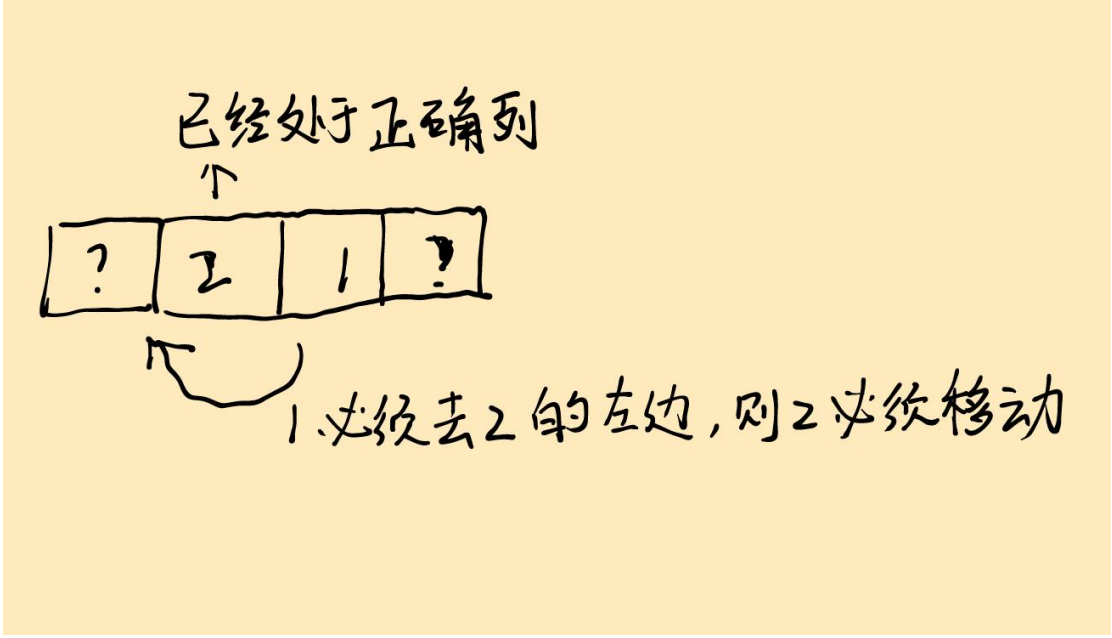
```
def wrongblocks(state:tuple):
    num = 1 #代表了某个位置上正确的数字
    count = 0 #用于统计错位的方块数量
    for i in range(4): #遍历 pullze 的每一个位置
        for j in range(4):
            if state[i][j] != 0 and state[i][j] != num: #如果该位置不等于 0，且不为正确的数字则错位的方块数加一
                count += 1
            num += 1
    return count
```

曼哈顿距离：曼哈顿距离是当前位置和目标位置的横纵距离差值之和，而另一种启发式函数则可以采用 1-15 的总曼哈顿距离。这种启发式函数的优点较第一个启发式的优点很明显，可以描绘出每个点到目标位置需要移动的步数，同时这种启发式函数是具有单调性的，每个点到目标位置的步数之和肯定比实际上把所有点全部归位的步数要少的多。

```
def manhatton(state:tuple):
    count = 0 #用于计算所有方块的曼哈顿距离之和
    for i in range(4): #遍历所有方块
        for j in range(4):
            num = state[i][j]
            right_i = (num-1) // 4 #正确的行
            right_j = (num-1) % 4 #正确的列
            if num != 0:
                count += abs(right_i - i) + abs(right_j - j) #计算曼哈顿距离
```

```
    return count
```

曼哈顿距离+线性冲突：所谓的线性冲突法，就是一个格子在移向它目标位置的沿途中，对沿途格子的影响。举个例子：



当 2 已经移到了它的目标位置，但是它右边的 1 的目标位置在它的左边，1 想要移到它的目标位置的话，要不就得从下面走，要不就得让 2 给它让路，这样 2 的位置则又乱了。所以这种情况就说明单单采用总的曼哈顿距离是不够准确的，加上线性冲突法能使启发式函数更接近于实际要走的步数。

采用这种方法能很大程度的提高 A*扩展点的效率。

然而，值得注意的是该启发式虽然是可接纳的，但是却不是单调的，这意味着该启发式可能并不能得到最优解。代码如下：

```
def linearconflict(state:tuple):
    count = 0
    for i in range(4):
        for j in range(4):
            num = state[i][j]
            right_i = (num-1) // 4
            right_j = (num-1) % 4
            if num != 0:
                count += abs(right_i - i) + abs(right_j - j) #与曼哈顿启发式相同

                if j == right_j: #如果当前方块在正确的列上
                    for k in range(j+1,4): #遍历在他右边的方块
                        if state[i][k] != 0 and (state[i][k]-1) // 4 == i and (state[i][k]-1) % 4 < j: #存在一个不为 0 的方块在他右边，但是正确位置在左边
```

```
                            count+=2 #该方块至少需要先移开一步再回来，所以至少需要加二
                        break
                if i == right_i: #如果该方块在正确的行上
                    for k in range(i+1,4): #遍历在他下面的方块
                        if state[k][j] != 0 and (state[k][j] -1) // 4 < i and (state[k][i]-1) % 4 == j: #存在一个不为 0 的方块在他下面，但是正确位置在他上面
                            count+=2 #同理加二
                        break
    return count
```

新的启发式函数：无论是采用曼哈顿距离还是曼哈顿距离加线性冲突做启发式函数，始终很难在有限的内存下解决测试 7 和测试 8，所以不得不大胆的采用一些别的启发式函数，通过扩大启发式函数的取值范围来排除许多不需要扩展的节点。通过查阅资料，我了解到这这样一个启发式函数如下：

f（n） = g(n)+h(n)+w(n)+8*s(n)

其中，g(n)为拓展节点的深度，h(n)为曼哈顿距离，w(n）为错位的方块数，s(n)的大小代表了 pullze 的混乱程度，具体为：首先令 s=0，遍历 pullze 中的数字，若前一个数字加一再对 16 取模不等于后一个数字，则 s 加一，遍历结束后返回 s。

通过实践，发现该启发式函数并不是单调的，也就是不一定能获得最优解，但是在扩展节点数较多，扩展深度较深时，此启发式函数能快速得到解，也不失为一种解决问题的方法。

代码如下：

```
def wrongextent(state:tuple):
    count = 0
    num1 = state[0][0]
    for i in range(4):
        for j in range(4):
            if (num1+1) % 16 != state[i][j] and (i != 0 or j != 0):
                count += 1
    return count
def my_heuristic(state:tuple):
    return manhatton(state)+wrongblocks(state)+8*wrongextent(state)
```

- **搜索算法**

A*：

```
def A_star():
    global extened_nodes
    open = queue.PriorityQueue() #建立优先队列 open 表
    closed  = dict() #建立集合 closed 表
    closed[pullze] = 0 #将初始 pullze 加入 closed 表中
    state = (pullze,find_zero_pos(pullze),0) #建立初始状态
```

```
    open.put((0,state)) #将 f(n)和初始状态加入 open 表
    parent = {} #用于存下路径
    actions = [] #用于存下每一步的动作
    steps_pullze = [] #用于存下每一步动作之后的 pullze
    while(not open.empty()):
        state = open.get()[1] #open 优先队列弹出一个状态
        if is_goal(state[0]): #判断是否到达目标状态
            break
        for i in range(4):
            move_block =
(state[1][0]+directions[i][0],state[1][1]+directions[i][1]) #计算得到需要
移动的方块的位置
            if is_vaild(move_block): #判断该位置是否合理，即坐标不能超过
pullze 的大小
                new_cost = closed[state[0]]+1 #得到新的 g(n)值
                new_pullze = change_to_list(state[0]) #将 pullze 改为 list，
因为 tuple 类型不允许修改，而这里需要交换放块的位置
                new_pullze[move_block[0]][move_block[1]],new_pullze[stat
e[1][0]][state[1][1]] =
new_pullze[state[1][0]][state[1][1]],new_pullze[move_block[0]][move_blo
ck[1]]
                new_pullze = change_to_tuple(new_pullze) #修改回 tuple 类型，
因为 list 类型不可哈希
                new_state = (new_pullze,move_block,i) #得到新的状态
                if ((not new_pullze in closed) or new_cost <
closed[new_pullze]): #如果新状态不在 closed 中
                    closed[new_pullze] = new_cost
                    open.put((new_cost+my_heuristic(new_pullze),new_stat
e)) #将 f(n)和新状态加入 open 表中
                    parent[new_pullze] = i #记录一下新状态来自的方向
                    extened_nodes += 1 #扩展节点数加一
                    if(extened_nodes % 10000 == 0):
                        print(extened_nodes)
    while(state[0] != pullze): #回溯得到路径，直到回溯到最初输入的 pullze
        steps_pullze.append(state[0]) #将每一步完成之后 pullze 记录下来
        direction = directions[parent[state[0]]]
        move_block = (state[1][0] - direction[0],state[1][1] -direction[1])
        new_pullze = change_to_list(state[0])
        actions.append(new_pullze[move_block[0]][move_block[1]]) #将每一
步的动作记录下来
        new_pullze[move_block[0]][move_block[1]],new_pullze[state[1][0]]
[state[1][1]] =
```

```
new_pullze[state[1][0]][state[1][1]],new_pullze[move_block[0]][move_blo
ck[1]]
        new_pullze = change_to_tuple(new_pullze)
        state = (new_pullze,move_block)
    actions.reverse() #reverse 记录的动作，使得动作正序输出
    steps_pullze.reverse()
    return actions,steps_pullze
```

我的 A*算法首先初始化开放列表和闭合列表，并将初始状态加入闭合列表和开放列表中。然后，在主循环中，它不断从开放列表中取出具有最小代价的状态进行扩展，直到找到目标状态为止。在每次状态扩展时，它会尝试四个方向上的移动，计算新状态的代价，并更新开放列表和闭合列表。一旦找到目标状态，它会从目标状态开始回溯，记录每一步的动作和状态，构建出最优路径。最终，它返回构建好的路径动作序列和状态序列。

IDA*：

```
def dfs(limit:int,actions:list,steps_pullze): #深度优先搜索
    global flag
    global goal_state
    global new_limit
    global extened_nodes
    state = steps_pullze[-1] #取上次递归的最后一个状态，主要作用为更新阈值后
能接着上次继续深度优先搜索
    if (flag == True): #如果已经找到目标状态就直接退出函数
        return actions,steps_pullze
    if(state[0] == goal): #判断是否找到目标状态
        goal_state = state
        flag =  True
        #print(closed[state[0]]+manhatton(state[0]))
        return actions,steps_pullze
    for i in range(4): #遍历上下左右四个方向
        move_block =
(state[1][0]+directions[i][0],state[1][1]+directions[i][1]) #通过计算值为
0 的方块的某个方向，得到需要移动方块的坐标
        if is_vaild(move_block): #如果该方块是有效的，即该方块的坐标大小不超
过 pullze 大小
            new_cost = closed[state[0]]+1 #新状态的 g(n)加一
            new_pullze = change_to_list(state[0])
            new_pullze[move_block[0]][move_block[1]],new_pullze[state[1]
[0]][state[1][1]] =
```

```
new_pullze[state[1][0]][state[1][1]],new_pullze[move_block[0]][move_blo
ck[1]]
            new_pullze = change_to_tuple(new_pullze)
            new_state = (new_pullze,move_block) #得到新状态
            if(new_cost+linearconflict(new_pullze) <= limit and ((not
new_pullze in closed) or new_cost<closed[state[0]])):
                closed[new_pullze] = new_cost #将新 pullze 加入 closed
                parent[new_state] = i
                extened_nodes += 1
                actions.append(state[0][move_block[0]][move_block[1]]) #
记录步骤
                steps_pullze.append(new_state) #记录每次完成步骤后的 pullze
                actions,steps_pullze = dfs(limit,actions,steps_pullze)
                if flag == False: #如果未找到目标状态，则需要弹出已经记录的步
骤和 pullze 等
                    actions.pop()
                    steps_pullze.pop()
                    del closed[new_pullze]
                else:
                    return actions,steps_pullze #否则，直接退出并返回步骤和
pullze
            elif(new_cost + linearconflict(new_pullze) > limit): #记录新的
阈值
                if new_cost+linearconflict(new_pullze) < new_limit:
                    new_limit = new_cost+linearconflict(new_pullze)
    return actions,steps_pullze
```

我的 IDA*算法通过迭代加深搜索的方式，在每轮迭代中逐渐增加搜索深度限制，通过深度优先搜索来探索状态空间。在搜索过程中，算法通过动态调整阈值来控制搜索的深度，以便在保证搜索的完备性的同时，降低内存消耗。一旦找到目标状态，算法即可终止搜索，并返回找到的路径动作序列和状态序列。

4. 创新点&优化（如果有）

- **采用字典来代替列表：**最初我使用的是 list 作为 closed 的数据结构，通过 list 来完成环检测。然而通过查阅资料，并进行测试得知，dict 的查询性能是优于 list 的，因为 dict 是基于 hash 实现的，每次查询的时间复杂度为 O(1)。并且我把 dict 的 value 设置成该状态的 g(n)值，这样还节省了存储空间，所以之后的 closed 都用 dict 完成。
- **线性冲突启发式函数：**线性冲突是指两个拼图块在同一行或同一列上并且目标位置在彼此之间时所产生的冲突。对于每一行和每一列，计算其中存在的线性冲突数，并将其加倍加到曼哈顿距离中。这个启发式函数能够更好地反映拼图块之间的相互干扰。然而，通过查阅资料得知，这一启发式函数不是单调的，所以不一定会有最优解。

- 新的启发式函数：虽然该启发式函数使得算法不一定能得到最优解，但在复杂的情形下往往能得到一个解。

三、 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）

以下测试前四个测试为压缩包中的测试，后四个测试为实验指导书上的测试。由于测试 7，测试 8 测试时间较长，所以只使用优化后的代码测试。由于程序的运行时间受各方面影响，所以以下实验结果中的 used time 可能每一次测试都会不同，甚至有较大差异。

以下的测试只包含测试 1-6.测试 7，8 会进行额外讨论。由于新的启发式函数不一定能得到最优解，并且曼哈顿距离或线性冲突足够解决前六个测试，所以测试 1-6 不采用新的启发式函数。

A*：

曼哈顿距离

测试 1：

```
-------step40-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.07051873207092285s
steps: [6, 11, 4, 14, 5, 8, 9, 5, 8, 3, 2, 6, 14, 8, 15, 7, 10, 4, 8, 15, 3, 2, 6, 14, 15, 10, 7, 3, 2, 6, 14, 15, 10, 7, 3, 2, 6, 10, 11, 12]
 total steps: 40
 extended nodes: 7783
```

测试 2：

```
-------step40-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.22271251678466797s
steps: [15, 14, 4, 2, 12, 15, 14, 9, 3, 5, 11, 13, 5, 14, 2, 12, 15, 11, 14, 2, 9, 8, 10, 4, 8, 10, 7, 3, 2, 9, 10, 7, 3, 2, 6, 5, 9, 10, 11, 15]
 total steps: 40
 extended nodes: 8191
```

测试 3：

```
-------step40-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.011203289031982422s
steps: [11, 14, 9, 1, 12, 4, 1, 8, 2, 13, 15, 12, 8, 2, 3, 11, 14, 9, 6, 1, 2, 3, 11, 7, 13, 11, 7, 14, 10, 13, 14, 10, 9, 5, 1, 2, 3, 7, 11, 15]
 total steps: 40
 extended nodes: 1082
```

测试 4：

```
-------step40-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.21923446655273438s
steps: [1, 8, 4, 1, 5, 11, 15, 3, 13, 15, 3, 10, 1, 5, 8, 4, 2, 1, 7, 14, 1, 7, 5, 3, 10, 6, 14, 5, 7, 2, 3, 7, 6, 14, 9, 13, 14, 10, 11, 12]
 total steps: 40
 extended nodes: 22709
```

测试 5：

```
-------step49-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 10.502003908157349s
steps: [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]
 total steps: 49
 extended nodes: 809084
```

测试 6：

```
-------step48-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 122.21576976776123s
steps: [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]
 total steps: 48
 extended nodes: 4719429
```

IDA*：

曼哈顿距离：

测试 1：

```
-------step40-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.23788881301879883s
steps: [6, 11, 4, 14, 5, 8, 9, 5, 8, 3, 2, 6, 14, 8, 15, 7, 10, 4, 8, 15, 3, 2, 6, 14, 15, 10, 7, 3, 2, 6, 14, 15, 10, 7, 3, 2, 6, 10, 11, 12]
 total steps: 40
 extended nodes: 8179
```

测试 2：

```
-------step40-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.07062053680419922s
steps: [15, 14, 4, 2, 12, 15, 14, 8, 10, 4, 8, 9, 3, 5, 11, 13, 5, 14, 2, 12, 15, 11, 14, 2, 9, 10, 7, 3, 2, 9, 10, 7, 3, 2, 6, 5, 9, 10, 11, 15]
 total steps: 40
 extended nodes: 1797
```

测试 3：

```
-------step40-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.05201578140258789s
steps: [11, 14, 9, 1, 12, 4, 1, 8, 2, 13, 15, 12, 8, 2, 3, 11, 14, 9, 6, 1, 2, 3, 11, 7, 13, 11, 7, 14, 10, 13, 14, 10, 9, 5, 1, 2, 3, 7, 11, 15]
 total steps: 40
 extended nodes: 1656
```

测试 4：

```
-------step40-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.8065266609191895s
steps: [1, 8, 4, 1, 5, 11, 15, 3, 13, 15, 3, 10, 1, 5, 8, 4, 2, 1, 5, 3, 10, 5, 7, 14, 1, 2, 3, 7, 5, 6, 14, 5, 6, 14, 9, 13, 14, 10, 11, 12]
 total steps: 40
 extended nodes: 19278
```

测试 5：

```
-------step49-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 59.070279598236084s
steps: [6, 10, 9, 4, 14, 9, 4, 1, 10, 4, 1, 3, 2, 14, 9, 1, 3, 2, 13, 12, 14, 13, 5, 11, 8, 6, 4, 3, 2, 5, 12, 7, 11, 12, 7, 14, 13, 9, 5, 10, 6, 8, 12,
 7, 10, 6, 7, 11, 15]
 total steps: 49
 extended nodes: 1798426
```

测试 6：

```
-------step48-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 57.426595926284795
steps: [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]
 total steps: 48
 extended nodes: 853048
```

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）

A*

曼哈顿距离+线性冲突：

测试 1：

```
-------step40-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.0520014762878418s
steps: [6, 11, 4, 14, 5, 8, 9, 5, 8, 3, 2, 6, 14, 8, 15, 7, 10, 4, 8, 15, 3, 2, 6, 14, 15, 10, 7, 3, 2, 6, 14, 15, 10, 7, 3, 2, 6, 10, 11, 12]
 total steps: 40
 extended nodes: 4545
```

测试 2：

```
-------step42-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.12187647819519043s
steps: [15, 14, 4, 2, 12, 15, 14, 4, 10, 8, 4, 9, 3, 5, 11, 13, 5, 14, 2, 10, 8, 4, 7, 3, 9, 2, 10, 12, 15, 11, 14, 9, 2, 7, 3, 2, 6, 5, 9, 10, 11, 15
]
 total steps: 42
 extended nodes: 10824
```

测试 3：

```
-------step40-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.013007640838623047s
steps: [11, 14, 9, 1, 12, 4, 1, 8, 2, 13, 15, 12, 8, 2, 3, 11, 14, 9, 6, 1, 2, 3, 11, 7, 13, 11, 7, 14, 10, 13, 14, 10, 9, 5, 1, 2, 3, 7, 11, 15]
 total steps: 40
 extended nodes: 1082
```

测试 4：

```
-------step40-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.09056282043457031s
steps: [1, 8, 4, 1, 5, 11, 15, 3, 13, 15, 3, 10, 1, 5, 8, 4, 2, 1, 7, 14, 1, 7, 5, 3, 10, 6, 14, 5, 7, 2, 3, 7, 6, 14, 9, 13, 14, 10, 11, 12]
 total steps: 40
 extended nodes: 7251
```

测试 5：

```
-------step49-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 5.540430307388306s
steps: [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]
 total steps: 49
 extended nodes: 401729
```

测试 6：

```
-------step48-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 46.86368465423584s
steps: [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]
 total steps: 48
 extended nodes: 2313938
```

IDA*

曼哈顿距离+线性冲突：

测试 1：

```
 -------step40-------
 1  2  3  4
 5  6  7  8
 9  10 11 12
 13 14 15 0
 Used time: 0.459907293319702l5s
 steps: [6, 11, 4, 14, 5, 8, 9, 5, 8, 3, 2, 6, 14, 8, 15, 7, 10, 4, 8, 15, 3, 2, 6, 14, 15, 10, 7, 3, 2, 6, 14, 15, 10, 7, 3, 2, 6, 10, 11, 12]
  total steps: 40
  extended nodes: 11562
```

测试 2：

```
-------step40-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 0.103516817092895 51s
steps: [15, 14, 4, 2, 12, 15, 14, 8, 10, 4, 8, 9, 3, 5, 11, 13, 5, 14, 2, 12, 15, 11, 14, 2, 9, 10, 7, 3, 2, 9, 10, 7, 3, 2, 6, 5, 9, 10, 11, 15]
 total steps: 40
 extended nodes: 1984
```

测试 3：

```
 -------step40-------
 1  2  3  4
 5  6  7  8
 9  10 11 12
 13 14 15 0
 Used time: 0.010998725891113281s
 steps: [11, 14, 9, 1, 12, 4, 1, 8, 2, 13, 15, 12, 8, 2, 3, 11, 14, 9, 6, 1, 2, 3, 11, 7, 13, 11, 7, 14, 10, 13, 14, 10, 9, 5, 1, 2, 3, 7, 11, 15]
  total steps: 40
  extended nodes: 1064
```

测试 4：

```
 -------step40-------
 1  2  3  4
 5  6  7  8
 9  10 11 12
 13 14 15 0
 Used time: 1.8666768074035645s
 steps: [1, 8, 4, 1, 5, 11, 15, 3, 13, 15, 3, 10, 1, 5, 8, 4, 2, 1, 5, 3, 10, 5, 7, 14, 1, 2, 3, 7, 5, 6, 14, 5, 6, 14, 9, 13, 14, 10, 11, 12]
  total steps: 40
  extended nodes: 30428
```

测试 5：

```
-------step55-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 21.234204530715942s
steps: [8, 11, 15, 7, 5, 3, 13, 5, 3, 1, 9, 4, 14, 10, 4, 14, 2, 13, 5, 12, 13, 5, 14, 9, 1, 3, 12, 14, 9, 1, 6, 4, 1, 2, 10, 1, 2, 10, 5, 9, 10, 6, 3
, 15, 7, 12, 15, 7, 11, 8, 4, 3, 7, 11, 12]
 total steps: 55
 extended nodes: 89388
```

测试 6：

```
 -------step48-------
 1  2  3  4
 5  6  7  8
 9  10 11 12
 13 14 15 0
 Used time: 171.11016178131104s
 steps: [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]
  total steps: 48
  extended nodes: 1747553
```

	测试 1	测试 2	测试 3	测试 4	测试 5	测试 6
优化前 A*(曼哈顿距离)	0.0705	0.2227	0.0112	0.2192	10.5020	122.2157
优化后 A*（线性冲突)	0.0520	非最优解	0.0130	0.0906	5.5404	46.8637
优化前 IDA*（曼哈顿距离)	0.2378	0.0706	0.0520	0.8065	59.0702	57.4265
优化后 IDA*（线性冲突)	0.4559	0.1035	0.0110	1.8667	非最优解	171.1101
最优程序	优化后 A*	优化前 IDA*	优化后 IDA*	优化后 A*	优化后 A*	优化后 A*

测试 7、8：
如果使用曼哈顿距离+线性冲突，则只有 A*算法能在有限的时间（小于 3 小时）和有限的空间内（小于 14000Mb）得到测试 7 的最优解，如下图所示：

```
-------step56-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 4555.4386844463501s
steps: [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]
 total steps: 56
 extended nodes: 21659192
```

所以考虑使用新的启发式函数(A*)：
测试 7：不是最优解，但仅仅 133s 就能给出一个解，运行时间几乎提升了 34 倍

```
-------step60-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 133.6168875694275s
steps: [13, 10, 5, 12, 14, 15, 9, 6, 3, 1, 8, 3, 4, 11, 1, 4, 3, 2, 7, 8, 4, 3, 2, 7, 10, 13, 12, 14, 6, 5, 13, 10, 8, 4, 3, 2, 5, 6, 15, 9, 11, 5, 6,
 13, 14, 15, 13, 11, 9, 13, 11, 14, 10, 12, 15, 11, 14, 10, 11, 15]
 total steps: 60
 extended nodes: 4885665
```

测试 8：最优解，仅用时 302s

```
-------step62-------
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
Used time: 302.97719955444336s
steps: [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]
 total steps: 62
 extended nodes: 7841528
```

四、 思考题

无

五、 参考资料

十五数码问题研究及实现 - 百度文库 (baidu.com)