

# E3\_22336216



中山大學  
SUN YAT-SEN UNIVERSITY

人工智能实验

## 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	DCS315	专业（方向）	计算机科学与技术（系统结构）
学号	22336216	姓名	陶宇卓

## 1 实验题目

实验三：使用A\*和IDA\*算法求15-puzzle问题

## 2 实验内容

一、算法原理

- A\*算法

**A\*算法**是一种常用的启发式搜索算法，通常用于解决路径规划问题，如在地图中找到最短路径。它结合了Dijkstra算法的广度优先搜索和代价最佳优先搜索的优点，通过评估每个节点的代价函数来选择下一个要扩展的节点，以在搜索过程中尽快找到最优解。

以下是A\*算法的主要原理：

**启发式函数 (Heuristic Function)：** A\*算法使用启发式函数来估计从当前节点到目标节点的最短路径的成本。这个启发式函数通常记作 $h(n)$ ，它能够快速计算出两个节点之间的预计距离。启发式函数应该是乐观的，即它不能高估两个节点之间的实际距离。在15-puzzle问题里，常用曼哈顿距离作为启发式函数。

**代价函数 (Cost Function)：** A\*算法使用一个代价函数 $f(n)$ 来评估每个节点 $n$ 的优先级，该函数通常定义为 $f(n) = g(n) + h(n)$ ，其中 $g(n)$ 是从起始节点到节点 $n$ 的实际代价， $h(n)$ 是从节点 $n$ 到目标节点的估计代价。因此， $f(n)$ 表示从起始节点到目标节点经过节点 $n$ 的总代价估计值。

**开启列表 (Open List)：** A\*算法使用一个开启列表来存储待扩展的节点，这些节点可能是搜索过程中的候选节点。初始时，只有起始节点在开启列表中。一般来说，使用优先队列作为开启列表。

**关闭列表 (Closed List)：** A\*算法还使用一个关闭列表来存储已经扩展过的节点，避免重复扩展相同的节点。

**搜索过程：** A\*算法的搜索过程如下：

1. 从开放列表中选择 $f(n)$ 值最小的节点 $n$ 进行扩展。
2. 如果节点 $n$ 是目标节点，则搜索结束，找到了最优解。
3. 否则，将节点 $n$ 从开启列表中移除，并将其加入关闭列表。
4. 对节点 $n$ 的每个邻居节点 $m$ 进行如下操作：
5. 如果节点 $m$ 已经在关闭列表中，则忽略它。
6. 如果节点 $m$ 不在开启列表中，则将节点 $m$ 加入开启列表，并计算其 $f(m)$ 值。
7. 重复以上步骤，直到找到目标节点或开启列表为空。

**最优解：**一旦找到目标节点，可以通过追踪每个节点的父节点指针，从目标节点回溯到起始节点，从而得到最优路径。

- IDA\*算法

**IDA算法 (Iterative Deepening A)** 是A算法的一种变体，它通过迭代加深搜索的方式，在内存占用上更加高效，同时能够保证找到最优解。IDA算法在某些情况下比A\*算法更加适用，特别是对于内存受限的情况。

以下是IDA\*算法的主要原理：

**启发式函数 (Heuristic Function)**：与A算法类似，IDA算法也使用启发式函数来估计从当前节点到目标节点的最短路径的成本。

**深度限制 (Depth Limit)**：IDA\*算法采用迭代加深搜索的策略，即在每一次搜索中限制搜索的最大深度。初始时，设定一个初始的深度限制，然后逐渐增加这个深度限制，直到找到解为止。

**搜索过程**：IDA\*算法的搜索过程如下：

1. 从初始节点开始，设置初始的深度限制为0。
2. 执行深度优先搜索，但是在搜索的过程中限制搜索深度，直到达到当前深度限制或者找到目标节点。
3. 如果找到目标节点，则搜索结束，返回最优解。
4. 如果搜索到达当前深度限制但没有找到目标节点，则增加深度限制，再次执行深度优先搜索。
5. 重复以上步骤，逐渐增加深度限制，直到找到目标节点为止。
6. 剪枝 (Pruning)：在IDA\*算法的搜索过程中，可以利用启发式函数来进行剪枝操作，即通过评估节点的代价函数来决定是否继续向下搜索。如果某个节点的代价函数超过当前深度限制加上该节点到目标节点的启发式估计值 ( $g+h>limit$ )，则可以剪枝，不再向下搜索该节点。

**最优解**：一旦找到目标节点，可以通过追踪每个节点的父节点指针，从目标节点回溯到起始节点，从而得到最优路径。

IDA算法的优点是它在搜索过程中只需要很少的内存空间，因为它不需要维护开启列表和关闭列表，而是通过深度优先搜索的方式进行搜索。但是，IDA算法的缺点是在某些情况下可能会重复搜索相同的节点，导致效率不高。

- 启发式函数的性质

**可采纳性 (Admissibility)**：可采纳性是指启发式函数 (heuristic function) 的性质，即启发式函数不能高估从当前节点到目标节点的最短路径的成本。换句话说，对于任何节点  $n$ ，其启发式函数  $h(n)$  应该小于或等于从节点  $n$  到目标节点的实际最短路径的成本。如果一个启发式函数是可采纳的，那么  $A^*$  算法就能够保证找到的路径是最优路径。

**一致性 (Consistency)**：一个启发式函数是一致的，如果对于任何相邻节点  $n$  和其后继节点  $m$ ，满足以下条件： $h(n) \leq c(n, m) + h(m)$ ，其中  $c(n, m)$  表示从节点  $n$  到节点  $m$  的实际代价。换句话说，启发式函数的估计值不应该高于通过从节点  $n$  到节点  $m$  的实际代价再加上从节点  $m$  到目标节点的最优路径的估计代价。一致性启发式函数也被称为具有单调性 (monotonic) 或满足三角不等式的启发式函数。

需要注意的是：15-puzzle 其实相当于图搜索，在图搜索里，所以如果启发式函数不满足一致性的话，是有可能得不到最优解的。如果启发式函数不满足一致性，即存在某些状态  $n$  和它的后继状态  $n'$ ，使得  $h(n) > c(n, n') + h(n')$ ，那么  $A^*$  算法就无法保证找到最优解。在这种情况下，算法可能会沿着一个代价更高的路径继续搜索，导致无法达到最优解，这个问题报告后面会提到。

- 针对 15-puzzle 的具体实现：

## A\*算法实现原理：

### 初始化：

1. 创建一个初始状态的拼图实例，将其放入优先队列。
2. 初始化一个空集合，用于存储已访问过的状态。
3. 初始化一个计数器，用于统计扩展的节点数。
4. 循环直到找到解决方案：
  - a. 从优先队列中取出一个拼图实例。
  - b. 如果该状态已经在已访问集合中，则忽略。
  - c. 如果当前状态是目标状态，则返回解决方案。
  - d. 将当前状态添加到已访问集合中。
  - e. 对当前状态进行扩展，生成所有可能的后继状态，并将它们加入优先队列。
5. 重复以上步骤，直到找到解决方案或者优先队列为空。

### 计算启发式函数：

对于每个状态，计算其启发式函数的值，通常是曼哈顿距离或错位方块数。

## IDA\*算法实现原理：

### 初始化：

- 一、创建一个初始状态的拼图实例。
- 二、初始化一个空集合，用于存储已访问过的状态。
- 三、初始化一个阈值，作为深度限制。
- 四、迭代加深搜索：
  - (一) 从初始状态开始，使用深度优先搜索的方式进行搜索，限制搜索的深度不超过当前阈值。
  - (二) 在搜索过程中，记录每个状态的深度和启发式函数的值。
  - (三) 如果找到目标状态，则返回解决方案。
  - (四) 如果当前搜索的深度超过了阈值，则增加阈值，再次进行搜索。

### 剪枝：

在搜索过程中，根据启发式函数的值进行剪枝操作，避免搜索不必要的状态。

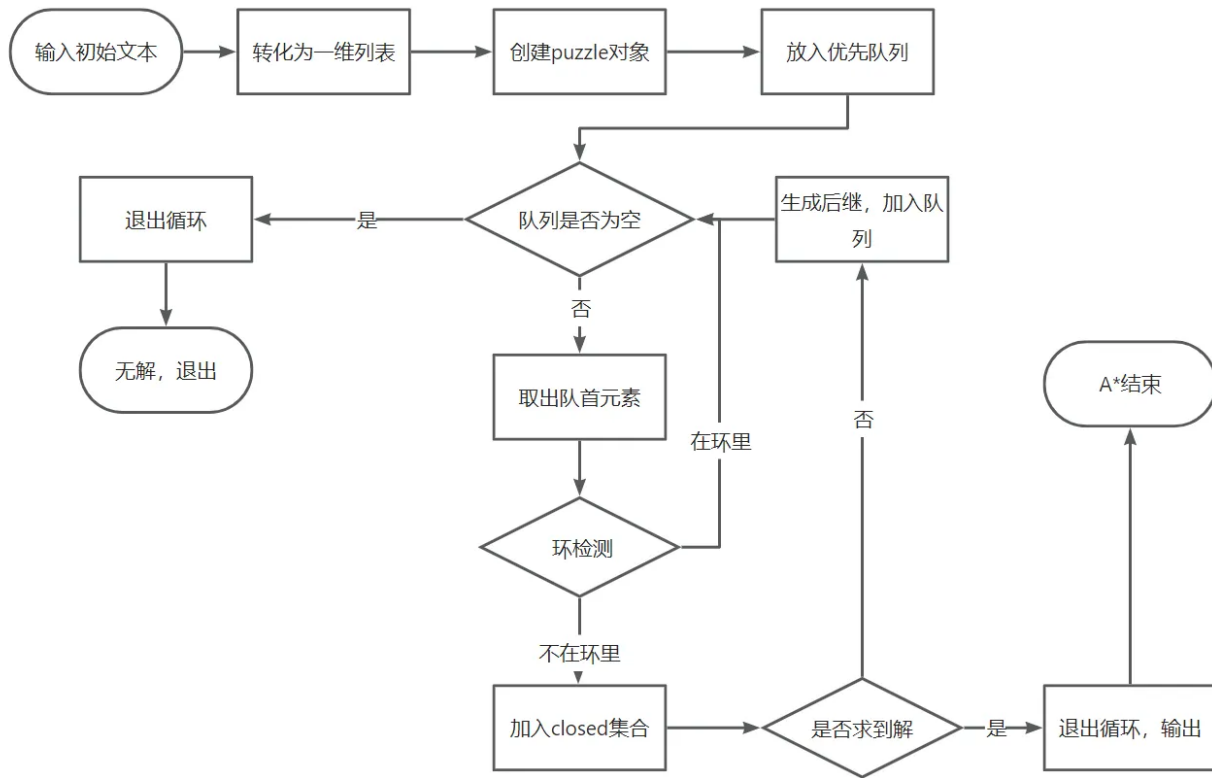
如果某个状态的启发式函数值超过了当前阈值，则不再扩展该状态。

### 重复迭代：

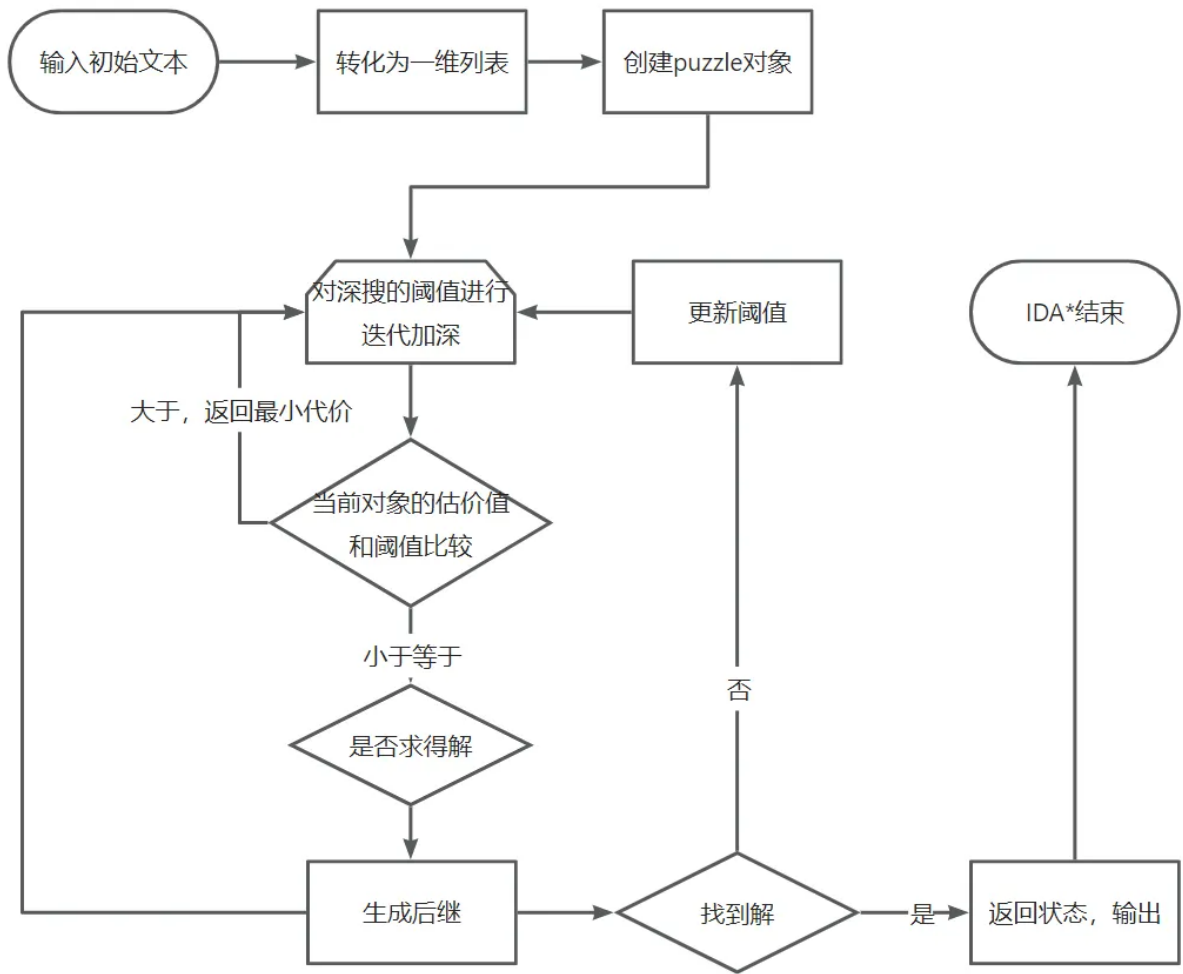
不断增加阈值，重复进行迭代加深搜索，直到找到解决方案为止。

- 流程图

- A\*算法



- IDA\*算法



## 二、伪代码

```
1  导入 copy、time、PriorityQueue 模块
2
3  定义 Puzzle 类:
4      初始化方法 __init__:
5          接受一个包含 16 个整数的列表 list16 作为参数
6          将列表转换为元组, 并赋值给属性 state
7          初始化路径 path 为空列表
8          初始化路径成本 g 为 0
9          初始化启发式值 h 为 0
10         计算空白格索引并赋值给属性 zero
11         调用 hn 方法计算启发式值
12
13     定义 __lt__ 方法:
14         接受另一个 Puzzle 实例 other、启发式值权重 h_mul 和路径成本权重 g_mu
15         l 作为参数
16         比较当前实例和另一个实例的 f 值, 如果当前实例的 f 值小于另一个实例的 f
17         值, 或者 f 值相等但 h 值小于另一个实例的 h 值, 则返回 True, 否则返回 False
18
19     定义 hn 方法:
20         计算当前状态的启发式值 h
21
22     定义 successors 方法:
23         接受一个集合 visited 作为参数
24         获取空白格的索引
25         根据空白格的位置生成所有可能的移动
26         遍历所有可能的移动:
27             创建一个新的状态实例, 更新路径成本、路径和状态
28             返回生成器, 生成所有后继状态的 Puzzle 实例
29
30 定义 A_star 函数:
31     接受一个 Puzzle 实例 p 作为参数
32     创建一个优先队列 pri
33     将初始状态 p 放入队列
34     创建一个集合 visited 来存储已访问过的状态
35     当队列不为空时循环:
36         从队列中获取一个状态 puz
37         如果当前状态已经访问过, 则跳过本次循环
38         如果当前状态的启发式值为 0, 则找到解决方案, 返回该状态
39         将当前状态添加到已访问集合中
40         遍历当前状态的所有后继状态:
41             将后继状态添加到队列中
```



```

42     接受一个 Puzzle 实例 p 作为参数
43     设置初始阈值为启发式值
44     创建一个集合 visited 来存储已访问过的状态
45     记录初始状态
46     循环：
47         调用 search 函数搜索解决方案
48         如果找到了解决方案，则返回解决方案
49         如果阈值为无穷大，则说明没有解决方案，返回初始状态
50
51 定义 search 函数：
52     接受当前状态 state、当前路径成本 g、深度限制 limit 和已访问集合 visited
    作为参数
53     计算当前状态的 f 值
54     如果 f 值大于阈值 limit，则返回当前状态和 f 值
55     如果当前状态的启发式值为 0，则找到解决方案，返回 'FOUND' 和当前状态
56     初始化最小值为无穷大
57     遍历当前状态的所有后继状态：
58         将后继状态添加到已访问集合中
59         递归搜索后继状态，并更新最小值
60         回溯，从已访问集合中移除当前状态
61     返回最小值和当前状态
62
63 定义 print_puzzle 函数：
64     接受一个列表 matrix 作为参数
65     遍历列表并打印拼图的状态
66
67 定义 move_blank 函数：
68     接受一个列表 matrix 和一个整数 move 作为参数
69     获取空白格和要移动的数字的索引，并交换它们的位置
70
71 定义主函数 main：
72     打开文件 'input7.txt' 并读取其中的内容到列表 lines
73     将列表 lines 转换为包含整数的列表 matrix
74     深度复制 matrix 并存储在 matrix1 中
75     创建 Puzzle 实例 a
76     记录开始时间
77     使用 A_star 算法解决拼图问题，并记录解决方案和运行时间
78     打印 A* 算法的解决方案和运行时间
79     打印解决方案中每一步的状态
80     记录结束时间
81     记录运行时间
82     使用 IDA_star 算法解决拼图问题，并记录解决方案和运行时间
83     打印 IDA* 算法的解决方案和运行时间
84     打印解决方案中每一步的状态
85
86

```

```
87  如果当前脚本为主程序：
88      调用主函数 main
```

### 三、关键代码展示

- 首先，封装一个Puzzle类，用于定义一个棋盘状态，内部包括状态state，初始状态到当前的移动路径path，h和g，还有棋盘内空白位置的索引，初始化对象的时候调用hn()函数，计算启发式函数值。successors()函数用于生成所有当前棋盘的后继状态。具体代码如下：

```

1  class Puzzle:
2      """
3      Puzzle类用于表示一个拼图的状态。
4      属性:
5          state: 一个元组，表示拼图的当前状态。
6          path: 一个列表，表示从初始状态到当前状态的路径。
7          g: 一个整数，表示从初始状态到当前状态的路径的成本。
8          h: 一个整数，表示当前状态的启发式值。
9          zero: 一个整数，表示空白格在拼图中的索引。
10     """
11
12     def __init__(self, list16):
13         """
14         初始化Puzzle类的一个新实例。
15         参数:
16             list16: 一个列表，包含16个整数，表示拼图的初始状态。
17             path: 一个列表，表示从初始状态到当前状态的路径。
18             g: 一个整数，表示从初始状态到当前状态的路径的成本。
19             h: 一个整数，表示当前状态的启发式值。
20             zero: 一个整数，表示空白格在拼图中的索引。
21         """
22         self.state = tuple(list16)  # 使用元组以便内存减少，所以状态应该是
不可变的
23         self.path = []  # 初始化路径为空
24         self.g = 0
25         self.h = 0
26         self.zero = self.state.index(0)  # 找到空白格的索引
27         self.hn()  # 计算启发式值
28
29     def __lt__(self, other, h_mul=1.0, g_mul=1.0):
30         """
31         定义两个Puzzle实例之间的比较方法。
32         参数:
33             other: 另一个Puzzle实例。
34             h_mul: 启发式值的权重。
35             g_mul: 路径成本的权重。
36
37         返回:
38             如果当前实例的f值小于另一个实例的f值，或者f值相等但h值小于另一个实
例的h值，返回True，否则返回False。
39         """
40         return g_mul * self.g + h_mul * self.h < g_mul * other.g + h_
mul * other.h or (

```

```

41         g_mul * self.g + h_mul * self.h == g_mul * other.g +
h_mul * other.h and self.h < other.h)
42
43     def hn(self):
44         """
45         计算当前状态的启发式值。
46         """
47         h1 = 0 # 曼哈顿
48         h2 = 0 # 线性冲突
49         for i in range(4):
50             for j in range(4):
51                 num = self.state[i * 4 + j]
52                 if num != 0:
53                     row_goal = (num - 1) // 4
54                     col_goal = (num - 1) % 4
55                     h1 += abs(i - row_goal) + abs(j - col_goal)
56                     if j == col_goal:
57                         for k in range(j + 1, 4):
58                             next_num = self.state[i * 4 + k]
59                             if next_num != 0 and (next_num - 1) // 4
== i and (next_num - 1) % 4 < j:
60                                 h2 += 1
61                     if i == row_goal:
62                         for k in range(i + 1, 4):
63                             next_num = self.state[k * 4 + j]
64                             if next_num != 0 and (next_num - 1) % 4 =
= j and (next_num - 1) // 4 < i:
65                                 h2 += 1
66
67         self.h = h1 + h2
68
69     def successors(self, visited):
70         """
71         生成当前状态的所有后继状态。
72         返回:
73             一个生成器, 生成所有后继状态的Puzzle实例。
74         """
75         zero = self.state.index(0)
76         x = zero % 4
77         y = zero // 4
78         move = []
79         if x > 0: move.append(-1)
80         if y > 0: move.append(-4)
81         if x < 3: move.append(1)
82         if y < 3: move.append(4)
83

```

```

84         for i in range(len(move)):
85             new = list(self.state)
86             new[zero], new[zero + move[i]] = new[zero + move[i]], new
87             # 交换
88             if tuple(new) in visited:
89                 continue
90             puz_child = Puzzle(new)
91             puz_child.g = self.g + 1
92             puz_child.path = self.path[:]
93             puz_child.path.append(new[zero])
94             yield puz_child

```

- 接下来是A\*搜索算法，具体代码如下：

▼ A\*
Python |

```

1  def A_star(p):
2      """
3      使用A*算法解决拼图问题。
4      参数:
5          p: 一个Puzzle实例，表示拼图的初始状态。
6      返回:
7          一个Puzzle实例，表示拼图的解决方案。
8      """
9      sum = 0
10     pri = PriorityQueue() # 创建一个优先队列
11     pri.put(p) # 将初始状态放入队列
12     visited = set() # 创建一个集合来存储已访问过的状态
13     while not pri.empty(): # 当队列不为空时
14         puz = pri.get() # 从队列中获取一个状态
15         if puz.state in visited: # 如果这个状态已经被访问过，就跳过
16             continue
17         sum += 1
18         if puz.h == 0: # 如果启发式值为0，说明找到了解决方案
19             print("Finding", sum, "nodes")
20             return puz
21         visited.add(puz.state) # 将当前状态添加到已访问集合中
22         for puz_child in puz.successors(visited): # 遍历当前状态的所有
23             # 后继状态
24             pri.put(puz_child) # 将后继状态添加到队列中

```

- 接下来是IDA\*算法，具体代码如下：

```

1  def IDA_star(p):
2      """
3      使用IDA*算法解决拼图问题。
4      参数:
5          p: 一个Puzzle实例, 表示拼图的初始状态。
6      返回:
7          一个列表, 包含从初始状态到解决方案的所有状态的Puzzle实例。
8      """
9      threshold_value = p.h # 设置初始阈值为启发式值
10     visited = set() # 创建一个集合来存储已访问过的状态
11     visited.add(p.state) # 记录初始状态
12     while True:
13         threshold_value, newstate = search(p, 0, threshold_value, visited)
14         if threshold_value == 'FOUND': # 如果找到了解决方案, 返回解决方案
15             return newstate
16         if threshold_value == float('inf'): # 如果阈值为无穷大, 说明没有
            解决方案
17             return p
18
19
20  def search(state, g, limit, visited):
21      """
22      使用深度优先搜索解决拼图问题。
23      参数:
24          state: 一个Puzzle实例, 表示当前状态。
25          g: 一个整数, 表示当前路径的成本。
26          limit: 一个整数, 表示搜索的深度限制。
27          visited: 一个集合, 包含已经访问过的所有状态。
28      返回:
29          如果找到解决方案, 返回 'FOUND'; 否则返回下一次搜索的深度限制。
30      """
31      f = g + state.h # 计算f值
32      if f > limit: # 如果f值大于阈值, 返回f值和当前状态
33          return f, state
34      if state.h == 0: # 如果启发式值为0, 说明找到了解决方案
35          return 'FOUND', state
36      min_val = float('inf') # 初始化最小值为无穷大
37      for successor in state.successors(visited): # 遍历当前状态的所有后
            继状态
38          visited.add(successor.state) # 将后继状态添加到已访问集合中
39          temp, newstate = search(successor, g + 1, limit, visited) #
            递归搜索后继状态

```

```

40     if temp == 'FOUND': # 如果找到了解决方案, 返回'FOUND'和解决方案
41         return 'FOUND', newstate
42     if temp < min_val: # 如果新的f值小于最小值, 更新最小值
43         min_val = temp
44         visited.remove(successor.state) # 回溯, 从已访问集合中移除当前状
    态
45     return min_val, state # 返回最小值和当前状态
46

```

- 主函数

```
1 def main():
2     """
3     主函数，从文件中读取拼图的初始状态，然后使用A*算法和IDA*算法解决拼图问题，并
    打印解决方案和运行时间。
4     """
5     # matrix = [0, 5, 15, 14, 7, 9, 6, 13, 1, 2, 12, 10, 8, 11, 4,
3] # example initial state
6     with open('input7.txt', 'r') as f:
7         lines = f.readlines()
8         matrix = [int(num) for line in lines for num in line.split()]
9         matrix1 = copy.deepcopy(matrix)
10        # print(matrix)
11        a = Puzzle(matrix)
12        time_start = time.time() # 计算时间
13        ans = A_star(a)
14        time_end = time.time()
15        time_c = time_end - time_start
16
17        print("A* Algorithm Solution:")
18        print("Total moves:", ans.g)
19        print('A*:', 'time cost', time_c, 's')
20        print(ans.path, end='\nstate:\n')
21        for move in ans.path: # 打印解决方案
22            print_puzzle(matrix1)
23            print("-----")
24            move_blank(matrix1, move)
25            print_puzzle(matrix1)
26            print("-----")
27            # print(ans.path)
28
29            time_start = time.time() # 计算时间
30            ans = IDA_star(a)
31            time_end = time.time()
32            time_c = time_end - time_start
33
34            print("\nIDA* Algorithm Solution:")
35            print("Total moves:", ans.g)
36            print('IDA*:', 'time cost', time_c, 's')
37            print(ans.path, end='\nstate:\n')
38            for move in ans.path: # 打印解决方案
39                print_puzzle(matrix)
40                print("-----")
41                move_blank(matrix, move)
```



```
42         print_puzzle(matrix)
43         print("-----")
44
45
46     if __name__ == '__main__':
47         main()
48
```

#### 四、创新点&优化

- 生成后继状态时提前剪枝

为了避免在这种问题中生成已在closed列表里的状态而多调用一次构造函数，我选择在**生成后继状态时剪枝而不是在生成完剪枝**，具体操作是在successors函数中提前检测newstate是否在传入的visited内，这样可以避免多调用初始化，切片等操作，有利于节约内存和时间。

```
if tuple(new) in visited:
    continue
```

- 采用set作为closed列表

集合是一种哈希表数据结构，具有快速的查找性能。集合的查找操作的平均时间复杂度是 $O(1)$ ，而列表一般是 $O(n)$ ，这是因为Python的集合是基于哈希表实现的，它使用哈希函数将元素映射到哈希表的特定位置。在搜索算法中，需要频繁地检查某个状态是否已经在闭列表中，使用集合可以在常量时间内完成查找操作，因此能够加速算法的执行速度。其次，闭列表用于存储已经访问过的状态，防止算法重复访问相同的状态。集合具有**自动去重**的特性，即使尝试将已经存在于集合中的状态再次添加进去，也不会造成重复元素的存在。这样可以避免算法陷入无限循环，同时减少不必要的状态扩展。这样可以节省内存空间，尤其是在存储大量状态时，能够有效减少内存占用。

- 改善启发式函数

我认为这个是最重要的优化，启发式函数的作用是指导搜索算法朝着最有可能导致解的方向前进。通过提供更准确的估计值，启发式函数可以使搜索算法更快地找到解决方案，从而大大提高搜索效率。同时，它可以帮助算法识别和优先考虑那些更有希望导致解的状态，从而减少需要探索的状态空间。这对于大规模问题尤其重要，因为状态空间可能非常庞大，使用一个好的启发式函数可以避免不必要的状态扩展，提高搜索效率。通过减少搜索算法的运行时间和内存占用，改善启发式函数可以使算法更快地找到解决方案，并减少计算资源的消耗。这对于在资源受限的环境下运行搜索算法的应用尤其重要。

我在网络上看到了一种改善以曼哈顿距离作为启发式函数的方法，叫做**线性冲突（linear conflict）**。线性冲突指的是在同一行或同一列上存在两个方块，它们的目标位置也在同一行或同一列上，并且它们的当前位置阻碍了彼此到达目标位置。这种情况下，由于这两个方块需要在同一行或同一列上移动到达目标位置，因此曼哈顿距离会低估实际的移动成本。线性冲突法通过将曼哈顿

距离与线性冲突的数量相结合来修正启发式函数。具体来说，对于每一对存在线性冲突的方块，将线性冲突的数量加到曼哈顿距离中。这样就可以更准确地估计移动的成本，从而改善了启发式函数的准确性。

但是需要注意的是，线性冲突+曼哈顿距离并没有一致性，也就是说这个启发式函数并不能保证对于所有15-puzzle问题都能得到最优解。因为在这种方法中，每次移动滑块最多能使h下降2，这样的话，会存在相邻节点n和其后继节点m，满足以下条件： $h(n) > c(n, m) + h(m) = 1 + h(m)$ 。这样就说明了这种方法不具备一致性。在之后的评测指标分析环节也可以看到提升这种方法内的线性冲突比例将会使程序并不能适用于所有样例。

在经过尝试之后，我选择曼哈顿距离加上线性冲突（每次冲突权值+1）作为启发式函数，经过测试，这个启发式函数虽然不满足一致性，但是针对大部分样例能够有效地提升程序运行时间。

- 

### 3 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）

图1为A\*，图二为IDA\*，列表为移动顺序。

- input1:

```
Finding 2209 nodes
A* Algorithm Solution:
Total moves: 40
A*: time cost 0.06934046745300293 s
[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]
```

```
IDA* Algorithm Solution:
Total moves: 40
IDA*: time cost 0.13874292373657227 s
[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]
```

- input2:

```
Finding 5553 nodes
A* Algorithm Solution:
Total moves: 40
A*: time cost 0.16838455200195312 s
[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]
```

```
IDA* Algorithm Solution:
Total moves: 40
IDA*: time cost 0.3154020309448242 s
[15, 14, 4, 15, 14, 9, 3, 5, 11, 13, 5, 14, 12, 2, 15, 12, 2, 11, 14, 2, 9, 8, 10, 4, 8, 10, 7, 3, 2, 9, 10, 7, 3, 2, 6, 5, 9, 10, 11, 15]
```

- input3:

```
Finding 226 nodes
A* Algorithm Solution:
Total moves: 40
A*: time cost 0.006061077117919922 s
[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]
```

```
IDA* Algorithm Solution:
Total moves: 40
IDA*: time cost 0.008163928985595703 s
[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]
```

- input4:

```
Finding 8372 nodes
A* Algorithm Solution:
Total moves: 40
A*: time cost 0.26169729232788086 s
[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]
```

```
IDA* Algorithm Solution:
Total moves: 40
IDA*: time cost 1.0599963665008545 s
[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]
```

- input5:

```
Finding 329598 nodes
A* Algorithm Solution:
Total moves: 49
A*: time cost 11.77354097366333 s
[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]
```

```
IDA* Algorithm Solution:
Total moves: 49
IDA*: time cost 32.21916055679321 s
[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]
```

- input6:

```
Finding 1145131 nodes
A* Algorithm Solution:
Total moves: 48
A*: time cost 42.25983166694641 s
[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]
```

```
IDA* Algorithm Solution:
Total moves: 48
IDA*: time cost 141.55839490890503 s
[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]
```

- input7:

```
Finding 23787036 nodes
A* Algorithm Solution:
Total moves: 56
A*: time cost 2029.2186286449432 s
[5, 12, 9, 10, 13, 5, 12, 13, 8, 2, 5, 8, 10, 6, 3, 1, 2, 3, 4, 11, 1, 4, 3, 2, 4, 3, 2, 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]
```

```
IDA* Algorithm Solution:
Total moves: 56
IDA*: time cost 4116.521168479767 s
[5, 12, 9, 6, 4, 15, 6, 10, 13, 5, 12, 13, 8, 4, 3, 1, 4, 2, 5, 8, 10, 6, 15, 11, 1, 3, 2, 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]
```

- input8:

```
Finding 13323461 nodes
A* Algorithm Solution:
Total moves: 62
A*: time cost 1451.2233798503876 s
[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]
```

```
IDA* Algorithm Solution:
Total moves: 62
IDA*: time cost 6343.228711284376 s
[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]
```

2. 评测指标展示及分析

• A\*算法（时间保留五位小数）

	A*时间（曼哈顿）	步数	搜索节点数	A*时间（曼哈顿+线性冲突1)	步数	搜索节点数	A*时间（曼哈顿+线性冲突2)	步数	搜索节点数
input1	0.07833s	40	3670	0.06934s	40	2209	0.09564s	40	2437
input 2	0.02870s	40	1427	0.15155s	40	5553	0.16612s	42	5426
input 3	0.01013s	40	548	0.00606s	40	226	0.00509s	40	158
input 4	0.26350s	40	11862	0.26170s	40	8372	0.15742s	40	4047
input 5	12.71984s	49	434355	11.77354s	49	329598	6.24134s	49	159770
input 6	74.13399s	48	2376248	42.25983s	48	1145131	26.11369s	48	636219
input 7	2873.47751s	56	28838241	2029.21863s	56	23787036	552.98602s	56	13735218
input 8	4822.43999	62	36027498	1451.22337s	62	25590878	627.20558s	62	13118377

• IDA\*算法（时间保留五位小数）

	IDA*时间（曼哈顿）	步数	IDA*时间（曼哈顿+线性冲突1)	步数	IDA*时间（曼哈顿+线性冲突2)	步数
input1	0.10445s	40	0.13874s	40	0.12122s	40
input2	0.07485s	40	0.31540s	40	0.89844s	40

input3	0.01559s	40	0.00816s	40	0.00508s	40
input4	1.14271s	40	1.05999s	40	0.88536s	40
input5	22.24013s	49	32.21916s	49	19.17863s	49
input6	235.77857s	48	141.55839s	48	26.11369s	48
input7	5784.55456s	56	4116.52116s	56	3757.64168s	56
input8	7794.54036s	62	6343.22071s	62	5778.87603s	62

分析以上结果，我们可以看到加上线性冲突之后确实会提升不少的速度，但是线性冲突权值2的优化方案不足以通过8个样例，权值1的可以恰好通过8个样例。


A\*算法的时间复杂度取决于搜索过程中的状态扩展次数，以及每次扩展状态的代价计算。在最坏情况下，A\*算法的时间复杂度是指数级的，即 $O(b^d)$ ，其中b是状态扩展的平均分支因子，d是最优解的深度。在实际情况下，由于使用了启发式函数，A\*算法通常能够减少搜索空间，因此实际运行时间可能会更短。

IDA\*算法的时间复杂度也取决于搜索过程中的状态扩展次数和每次扩展状态的代价计算。IDA\*算法是一种迭代加深搜索算法，在每次迭代中限制搜索的深度，直到找到解决方案为止。因此，IDA\*算法的时间复杂度是 $O(b^d)$ ，其中b和d的含义与A算法相同。

A\*算法的空间复杂度主要取决于维护的优先队列（PriorityQueue）和已访问状态的集合。优先队列用于存储待扩展的状态，其空间复杂度是 $O(b^d)$ ，其中b和d的含义与上述相同。已访问状态的集合用于记录已经访问过的状态，其空间复杂度也是 $O(b^d)$ 。

名称	14% CPU	96% 内存
 Python	10.2%	11,041.3 MB

IDA\*算法的空间复杂度主要取决于递归搜索过程中维护的递归调用栈和已访问状态的集合。递归调用栈的空间复杂度取决于搜索的深度，通常是 $O(d)$ ，其中d是搜索的最大深度。已访问状态的集合的空间复杂度也是 $O(b^d)$ 。

名称	14% CPU	30% 内存
>  PyCharm	0.3%	518.5 MB

## 4 思考题

无

## 5 参考资料

 [【人工智能】A\\*算法和IDA\\*算法求解15-puzzle问题（大量优化，能优化的基本都优化了）\\_十五数码问题a\\*-CSDN博客](#)