



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	刘咏梅老师班级	专业 (方向)	信息与计算科学
学号	22336313	姓名	郑鸿鑫

一、实验题目

使用 A*与 IDA*算法求 15-Puzzle 问题的最优解。

二、实验内容

1. 算法原理

A*算法原理:

A 算法是一种基于 Dijkstra 算法的启发式搜索算法,它使用一个优先队列(通常是最小堆)来选择下一步要探索的节点。A 算法的核心是评价函数 $f(n) = h(n) + g(n)$, 其中:

$g(n)$ 是从初始节点到当前节点 n 的实际代价或步数。

$h(n)$ 是当前节点 n 到目标节点的启发式估计代价,通常是基于问题领域的启发式函数来计算的。

$f(n)$ 是节点 n 的总估计代价。

A*算法通过维护两个集合: 开放集合 (open set) 和关闭集合 (closed set)。开放集合包含待评估的节点,而关闭集合包含已经评估过的节点。算法从将初始节点加入开放集合开始,然后不断迭代以下步骤:

从开放集合中选择 $f(n)$ 值最小的节点作为当前节点。

检查当前节点是否为目标节点。如果是,那么路径被找到;如果不是,继续下一步。

将当前节点从开放集合移动到关闭集合。

扩展当前节点,生成所有可能的后继节点,并为每个后继节点计算 $g(n)$ 和 $h(n)$ 。

如果后继节点已经在关闭集合中,忽略它;否则,将其添加到开放集合中,并记录其父节点。这个过程一直重复,直到找到目标节点或开放集合为空(表示无解)。

IDA*算法原理:

DA*(Iterative Deepening A*)算法是一种结合了 A 算法和迭代加深搜索(Iterative Deepening Search, IDS)的启发式搜索算法。IDA 算法旨在克服传统 A*算法在解决路径问题时可能遇到的两个主要问题: 内存消耗和时间复杂度。

1.启发式搜索: IDA 算法使用 A 算法中的启发式函数来评估每个节点的 f 值(总估计成本), 这个值由 g 值(从起点到当前节点的实际成本)和 h 值(启发式估计的从当前节点到目标的成本)之和计算得出。

2.迭代加深: IDA*算法通过迭代加深的方式逐步增加搜索深度的界限(limit)。在每次迭代中, 算法都会尝试找到一个解, 如果在这个深度限制下找不到解, 则增加深度限制并重新开始搜索。

3.节点存储: 在每次迭代中, 算法使用一个栈(stack)来存储待扩展的节点。每个节点包含其状态(即节点的矩阵表示)、当前的 f 值、g 值和动作序列(用于记录从初始状态到当前状态所采取的动作)。

4.深度限制: 算法开始时设定一个初始深度限制(limit), 并在每次迭代中逐渐增加这个限制。当找到一个解或达到深度限制时, 当前迭代停止。

5.节点扩展: 在每次迭代中, 算法会扩展栈中的节点, 直到找到目标状态或达到深度限制。如果节点的 f 值减去 g 值等于 0 (即已经达到目标状态), 则该节点被扩展并记录其路径。

6.更新深度限制: 如果当前迭代中没有找到解, 算法会更新深度限制为当前迭代中遇到的最小 f 值, 然后开始新的迭代。

7.终止条件: 当找到一个解时, 算法终止。如果所有可能的深度都已尝试过, 且没有找到解, 则算法终止并报告失败。

2. 伪代码

A*算法的伪代码: A*算法由以下两个函数 A_star 和 expand 实现, 伪代码如下:

```
Function A_star(matrix):  
    start_time = Get Current Time()  
    h = Calculate_h3(matrix)  
    matrix = Convert To Tuple(matrix)  
    open_list = [(h, matrix, (), 0)] // (Heuristic, State, Action Sequence, Steps)  
    Heapify(open_list)  
    closed_set = New Set()
```



Print "动作序列如下:"

While True:

 If Heuristic of open_list[0][1] Is 0:

 end_time = Get Current Time()

 result = open_list[0]

 Print Resulting Actions(result[2])

 Print "步数为: ", Length of result[2]

 Print "所花时间为: ", end_time - start_time, "秒"

 Return Time Taken By Algorithm

 expand(open_list, closed_set)

Function expand(open_list, closed_set):

 cur = Pop From Heap(open_list)

 closed_set.Add(cur[1])

 (i, j) = Find Position Of cur[1]

 neighbors = [(1, 0), (0, 1), (-1, 0), (0, -1)] // 可能的移动方向

 For Each neighbor In neighbors:

 If neighbor Is Out Of Bounds Or neighbor Is In closed_set:

 Continue

 temp = Get Element From cur[1] At neighbor

 new_matrix = Swap Elements In cur[1] At (i, j) With neighbor

 If new_matrix Is In closed_set:

 Continue

 Else:

 h_new = Calculate_h3(new_matrix)

 step = cur[3] + 1

 Push Onto Heap(open_list, (h_new + step, new_matrix, result[2] + (temp,), step))

IDA*算法伪代码，IDA*算法主要由以下函数 IDA_star 实现，伪代码如下：

Function IDA_star(matrix):

 matrix = Convert To Tuple(matrix)

 start_time = Get Current Time()

 limit = Calculate_h3(matrix)

 stack = Empty Deque()

 flag = False

 closed_set = Empty Set()

 While True:

 stack.Append((matrix, limit, 0, ()))

 limi = Infinity

 closed_set.Clear()

 closed_set.Add(matrix)

 While stack Is Not Empty:

 cur = stack.Pop()

 If cur[1] - cur[2] Is 0:

 new_path = cur[3]

 flag = True



```
end_time = Get Current Time()
Break
Else:
    position = Find Position Of cur[0] Starting From 0
    (i, j) = position
    neighbors = [(1, 0), (0, 1), (-1, 0), (0, -1)]
    For Each neighbor In neighbors:
        If neighbor Is Out Of Bounds:
            Continue
        temp = Get Element From cur[0] At neighbor
        m_new = Swap Elements In cur[0] At (i, j) With neighbor
        If m_new Is In closed_set:
            Continue
        h = Calculate_h3(m_new)
        g = cur[2] + 1
        f = g + h
        If f <= limit:
            stack.Append((m_new, f, g, cur[3] + (temp,)))
            closed_set.Add(m_new)
        Else:
            limi = Min(limi, f)
    closed_set.Remove(cur[0])
If flag Is True:
    Break
Else:
    limit = limi
Print "动作序列为"
curr = matrix
For Each i In new_path:
    Print i, End With Space
Print()
Print "步数为:", Length of new_path
Print "所花时间为:", end_time - start_time, "秒"
```

3. 关键代码展示（带注释）

启发式函数 1：错牌数量

```
def calculate_h1(matrix): # 启发函数1：错牌数量
    h = 0
    for i in range(4):
        for j in range(4):
            if matrix[i][j] != 0 and matrix[i][j] != end[i][j]:
                h += 1
    return h
```

启发式函数 2：曼哈顿距离



```
def calculate_h2(matrix): # 启发式函数2: 曼哈顿距离
    distance = 0
    for i in range(4):
        for j in range(4):
            if matrix[i][j] != 0:
                target_row = (matrix[i][j]-1) // 4
                target_col = (matrix[i][j]-1) % 4
                distance += abs(i-target_row) + abs(j-target_col)
    return distance
```

启发式函数3: 结合线性冲突的曼哈顿距离

```
def calculate_h3(matrix): # 启发式函数3: 结合线性冲突的曼哈顿距离
    distance = 0
    linear_conflict = 0
    for i in range(4):
        for j in range(4):
            if matrix[i][j] != 0:
                target_row = (matrix[i][j] - 1) // 4
                target_col = (matrix[i][j] - 1) % 4
                distance += abs(i - target_row) + abs(j -
target_col)

                if i == target_row: # 在同一行
                    for k in range(j + 1, 4):
                        if matrix[i][k] != 0 and (matrix[i][k] - 1)
// 4 == target_row and matrix[i][j] > matrix[i][k]:
                            linear_conflict += 1
                if j == target_col: # 在同一列
                    for k in range(i + 1, 4):
                        if matrix[k][j] != 0 and (matrix[k][j] - 1) %
4 == target_col and matrix[i][j] > matrix[k][j]:
                            linear_conflict += 1
    return distance + 2 * linear_conflict
```

find_x 函数: 传入两个参数, 一个为矩阵一个为数字 x, 返回 x 在矩阵中的坐标 (i, j):

```
def find_x(matrix,x):
    for i in range(4):
        for j in range(4):
            if matrix[i][j] == x:
                return (i,j)#寻找 x 的位置并返回
```

交换函数: 传入 3 个参数, 一个为原矩阵用元组保存, 后两个为需要执行交换的位置, 返回得到的矩阵:

```
def swap_elements (original, pos1, pos2):
    # 创建一个新的元组列表, 避免修改原始元组
    new_tuple_list = [None] * Len(original)
    for i, row in enumerate(original):
        new_tuple_list[i] = [None] * Len(row)
        for j, element in enumerate(row):
            new_tuple_list[i][j] = element
    # 交换元素
    new_tuple_list[pos1[0]][pos1[1]], new_tuple_list[pos2[0]][pos2[1]] =
new_tuple_list[pos2[0]][pos2[1]], new_tuple_list[pos1[0]][pos1[1]]
    # 将新的元组列表转换为元组的元组
    new_tuple = tuple(tuple(row) for row in new_tuple_list)
    return new_tuple
```



打印函数，用于打印每一步的矩阵形态：

```
def print_matrix(curr, path):
    for i in path:
        print("将0与", i, "交换得: ", sep = '')
        position = find_x(curr, 0)
        i1 = position[0]
        j1 = position[1]
        position = find_x(curr, i)
        i2 = position[0]
        j2 = position[1]
        curr = swap_elements(curr, (i1, j1), (i2, j2))
        print("-----")
        for m in range(4):
            for n in range(4):
                print(f"{str(curr[m][n]):<3}", end=' ')
            print()
```

A*算法函数，包含 A_star 和 expand 函数，expand 用于每次维护扩展后的 open 和 close 集合：

```
def expand(open:heapq, close:set):
    cur = heapq.heappop(open)
    close.add(cur[1])
    position = find_x(cur[1], 0)
    i = position[0]
    j = position[1]
    neighbors = [(i+1, j), (i, j+1), (i-1, j), (i, j-1)]
    for neighbor in neighbors:
        if neighbor[0] < 0 or neighbor[0] > 3 or neighbor[1] < 0 or
neighbor[1] > 3: #超出边界的节点不扩展
            continue
        temp = cur[1][neighbor[0]][neighbor[1]]
        m_new = swap_elements(cur[1], (i, j), neighbor)
        if m_new in close: #已经访问过的也跳过（环检测）
            continue
        else:
            h = calculate_h3(m_new)
            step = cur[3]+1
            heapq.heappush(open, (h+step, m_new, cur[2]+(temp, ), step))

def A_star(matrix):
    start = time.time()
    h = calculate_h3(matrix)
    matrix = tuple(map(tuple, matrix))
    open = [(h, matrix, tuple(), 0)] # 0 为启发值 1 为矩阵 2 为动作序列 3 为 g 值
    heapq.heapify(open)
    close = set()
    print("动作序列如下:")
    while True:
        if calculate_h3(open[0][1]) == 0:
            end = time.time()
            temp = open[0]
            curr = matrix
            #print_matrix(curr, temp[2])
            for i in temp[2]:
                print(i, end = ' ')
            print()
```



```
print("步数为: ", len(temp[2]))
print("所花时间为: ", end - start, "s", sep='')
return time
expand(open, close)
```

IDA*算法实现函数，进行深度受限的迭代 A*算法：

```
def IDA_star(matrix):
    matrix = tuple(map(tuple, matrix))
    start = time.time()
    limit = calculate_h3(matrix)
    stack = deque()
    flag = False
    while True:
        stack.append((matrix, limit, 0, tuple())) # 0 为节点状态 1 为当前 f 值 2
        # 为 g 值 3 为动作序列
        limi = float('inf')
        while len(stack) != 0:
            cur = stack.pop()
            if cur[1] - cur[2] == 0:
                new_path = cur[3]
                flag = True
                end = time.time()
                break
            else:
                position = find_x(cur[0], 0)
                i = position[0]
                j = position[1]
                neighbors = [(i+1, j), (i, j+1), (i-1, j), (i, j-1)]
                for neighbor in neighbors:
                    if neighbor[0] < 0 or neighbor[0] > 3 or neighbor[1]
                    < 0 or neighbor[1] > 3:
                        continue
                    temp = cur[0][neighbor[0]][neighbor[1]]
                    m_new = swap_elements(cur[0], (i, j), neighbor)
                    if len(cur[3]) != 0:
                        if temp == cur[3][-1]: # 进行路径检测，避免产生逆操作
                            continue
                    h = calculate_h3(m_new)
                    g = cur[2] + 1
                    f = g + h
                    if f <= limit:
                        stack.append((m_new, f, g, cur[3] + (temp,)))
                    else:
                        limi = min(limi, f)
                if flag == True:
                    break
            else:
                limit = limi
        print("动作序列为")
        curr = matrix
        # print_matrix(curr, new_path)
        for i in new_path:
            print(i, end = ' ')
        print()
```

```
print("步数为:", len(new_path))  
print("所花时间为:", end - start, 's', sep='')
```

4. 创新点&优化

优化过程:

1.0 版本:

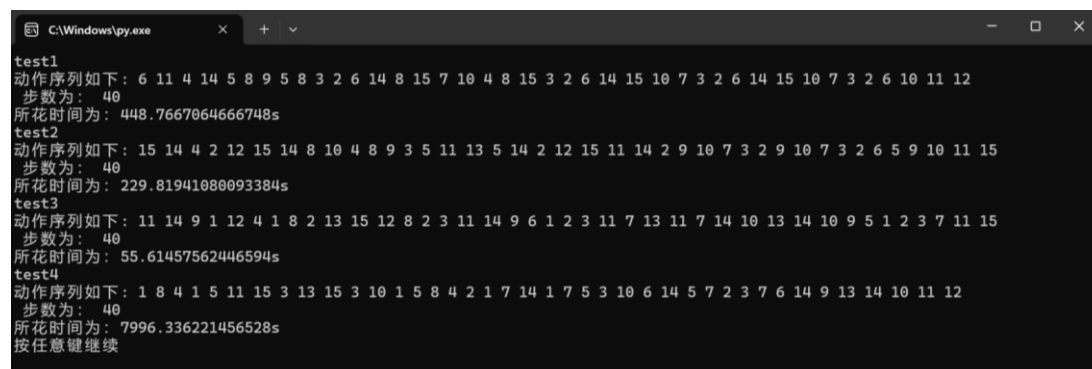
所有数据结构均采用列表实现, 使用错牌数量作为启发式函数, 该启发式函数效率较低下, 只能解决一些自己编写的 10 步之内的样例, 对于压缩包和 pdf 中的样例耗时太长, 故在此不放出代码详细介绍, 且在后续的实验中, 都不采用错牌数作为启发式函数。

2.0 版本:

所有数据结构均采用列表实现, 使用曼哈顿距离作为启发式函数, 由于列表占用内存较多, 且每次需要调用 sort 函数进行对 open 列表的排序, 当 open 和 close 的元素数量增加时, 排序效率低。

源代码可以在 code 文件夹中 15puzzle1.py 中找到。

对于压缩包的 4 个测试用例用 A*算法结果如下:



```
C:\Windows\py.exe  
test1  
动作序列如下: 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  
步数为: 40  
所花时间为: 448.7667064666748s  
test2  
动作序列如下: 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  
步数为: 40  
所花时间为: 229.81941080093384s  
test3  
动作序列如下: 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  
步数为: 40  
所花时间为: 55.61457562446594s  
test4  
动作序列如下: 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  
步数为: 40  
所花时间为: 7996.336221456528s  
按任意键继续
```

可以看到虽然都可以得出正确结果, 但是耗时非常的长, 而且对于 pdf 上的 4 个更加复杂的用例耗时更久, 所以后续 IDA*也不会使用列表这种数据结构。

2.1 版本

进行如下优化: 将 close 换成 set 集合来存储, 可以在判断某节点是否存在 close 中时无需遍历整个 close, 用堆来存储 open, 使得每次无需再对 open 进行排序, 这两点优化可以节省大量的运算时间。

由于每次优化时都是在原代码基础上修改, 对代码迭代时没有对代码进行存档, 故此处无法给出这一版本的原代码, 但是有保留测试结果。



```
zip_test1(优化后):  
动作序列如下: 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  
步数为: 40  
所花时间为: 1.9141509532928467s  
zip_test2(优化后):  
动作序列如下: 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  
步数为: 40  
所花时间为: 2.3454465866088867s  
zip_test3(优化后):  
动作序列如下: 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  
步数为: 40  
所花时间为: 0.05904054641723633s  
zip_test4(优化后):  
动作序列如下: 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  
步数为: 40  
所花时间为: 36.120689153671265s  
按任意键继续...
```

参考测试结果我们可以看出,在更换了数据结构后,算法的速度确实提高了几十甚至上百倍。

2.2 版本:

在此基础上我们加上 IDA*算法并且采用元组保存所有数据结构,根据搜索到的资料显示,元组会比列表在空间上和时间上更加节省,唯一的缺点是不可修改。

遗憾的是该版本的源代码依旧没有保存下来。

这里给出此版本代码下的运行结果:

```
zip_test1(A*):  
动作序列如下:  
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  
步数为: 40  
所花时间为: 1.4791693687438965s  
zip_test2(A*):  
动作序列如下:  
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  
步数为: 40  
所花时间为: 1.6715562343597412s  
zip_test3(A*):  
动作序列如下:  
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  
步数为: 40  
所花时间为: 0.03679680824279785s  
zip_test4(A*):  
动作序列如下:  
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  
步数为: 40  
所花时间为: 23.709944009780884s  
zip_test1(迭代加深A*)  
动作序列为  
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  
步数为: 40  
所花时间为: 0.5557043552398682s  
zip_test2(迭代加深A*)  
动作序列为  
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  
步数为: 40  
所花时间为: 0.27236318588256836s  
zip_test3(迭代加深A*)  
动作序列为  
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  
步数为: 40  
所花时间为: 0.04904317855834961s  
zip_test4(迭代加深A*)  
动作序列为  
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  
步数为: 40  
所花时间为: 43.82303500175476s  
按任意键继续...
```

可以观察到除了第 4 个例子外,其余所有样例的速度已经非常快了,再接着对 pdf 上的例子进行测试,结果如下,以 pdf 上第一个例子为例(版本修改较多,找不到此版本 A*算法对 pdf 样例的测试,故以 IDA*为例)



pdf_test1(迭代加深A*)

动作序列为: 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
步数为: 49
所花时间为: 4882.869408369064s

可以看到效率十分低下，检查代码发现，仍存在使用列表而不是使用元组的代码：

```
m_new = np.copy(cur[0])
temp = m_new[neighbor[0]][neighbor[1]]
m_new[neighbor[0]][neighbor[1]] = 0
m_new[i][j] = temp
```

2.3 版本:

此版本源代码可以在 code 文件夹中 15puzzle2.py 中找到。

针对 2.2 版本最后提到的代码段进行修改，但由于换成元组后不可修改，故采用新建一个元组来存储交换后的矩阵，并封装为函数：

```
temp = cur[1][neighbor[0]][neighbor[1]]
m_new = swap_elements(cur[1], pos1: (i,j), neighbor)
```

该函数的详细代码可以查阅上文关键代码展示中的交换函数，依靠此我们避免调用 np.copy() 函数，再次对 pdf 样例 1 进行测试，得到结果：

pdf_test1(迭代加深A*)

动作序列为: 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
步数为: 49
所花时间为: 1475.9620444774628s

可以看见效率提升了接近 4 倍，再对压缩包的样例进行测试，对比前版本，效率也有所提高，结果如下：

```
zip_test1(优化后):
动作序列如下:
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
步数为: 40
所花时间为: 1.214181661605835s
zip_test2(优化后):
动作序列如下:
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
步数为: 40
所花时间为: 1.388810157775879s
zip_test3(优化后):
动作序列如下:
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
步数为: 40
所花时间为: 0.02694559097290039s
zip_test4(优化后):
动作序列如下:
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
步数为: 40
所花时间为: 13.410313129425049s
zip_test1(迭代加深A*)
动作序列为
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
步数为: 40
所花时间为: 0.5140678882598877s
zip_test2(迭代加深A*)
动作序列为
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
步数为: 40
所花时间为: 0.2455754280090332s
zip_test3(迭代加深A*)
动作序列为
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
步数为: 40
所花时间为: 0.04124259948730469s
zip_test4(迭代加深A*)
动作序列为
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
步数为: 40
所花时间为: 40.55559587478638s
按任意键继续...
```

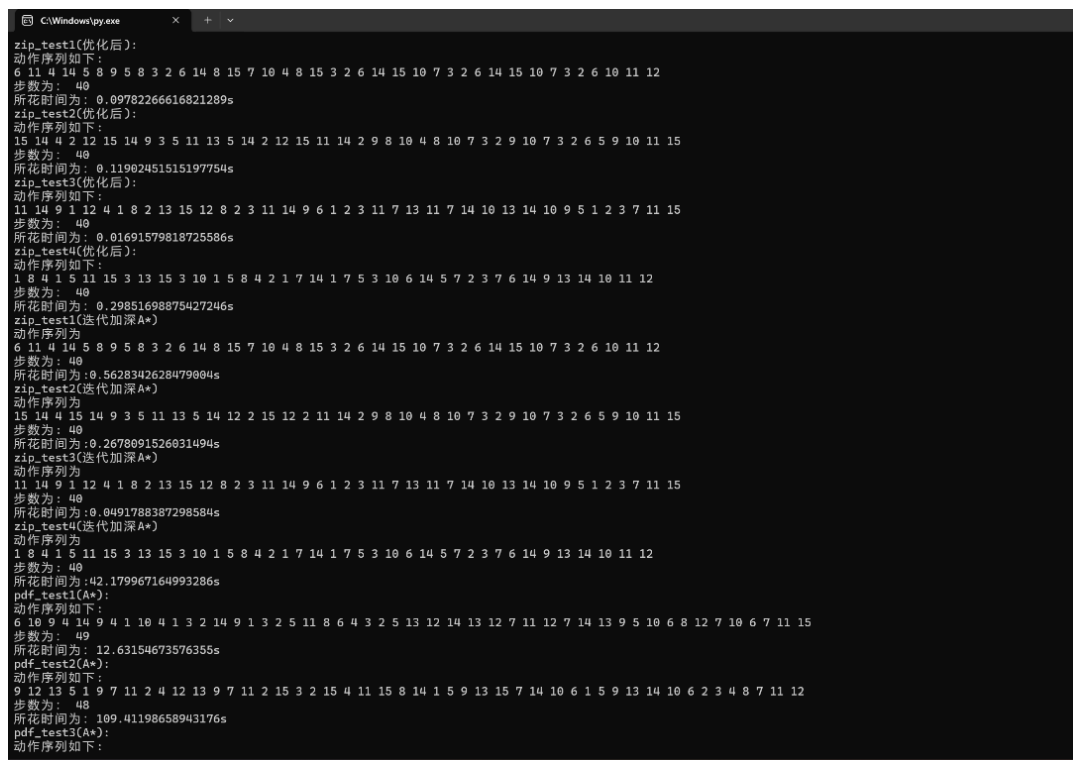
至此对压缩包的样例都已经十分高效，但是对于 pdf 样例耗时还是很长，A*算法甚至效率不如 IDA*，这点十分不符合常理，因为在相同启发式函数下，A*的速度应该是比 IDA*要快。经过询问两位助教，史哲源学长为我指出问题所在，在 2.3 版本的 A*算法中，下列代码通过回溯去复现搜索路径，需要进行对 close 集合的遍历，这在节点数目多的情况下是十分耗时的。故我参考 IDA*的写法，维护一个元组来保存路径，而不用通过回溯得到。

```
temp = open[0]
    action = []
    while True:
        parent = temp[4]
        action.append(temp[2])
        if temp[2] == ':' or temp[4] == None:
            break
        for i in close:
            if i[3] == parent:
                temp = i
                break
        action.reverse()
```

2.4 版本:

此版本源代码可以在 code 文件夹中 15puzzle3.py 中找到。

通过上述修改得到全新的 A*算法函数，对各样例测试，得到结果：



```
C:\Windows\py.exe
zip_test1(优化后):
动作序列如下:
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
步数为: 40
所花时间为: 0.09782266616821289s
zip_test2(优化后):
动作序列如下:
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
步数为: 40
所花时间为: 0.11902451515197754s
zip_test3(优化后):
动作序列如下:
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
步数为: 40
所花时间为: 0.01691579818725586s
zip_test4(优化后):
动作序列如下:
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
步数为: 40
所花时间为: 0.29851698875427246s
zip_test1(迭代加深A*):
动作序列为:
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
步数为: 40
所花时间为: 0.562834262847904s
zip_test2(迭代加深A*):
动作序列为:
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
步数为: 40
所花时间为: 0.2678091526031494s
zip_test3(迭代加深A*):
动作序列为:
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
步数为: 40
所花时间为: 0.0491788387298584s
zip_test4(迭代加深A*):
动作序列为:
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
步数为: 40
所花时间为: 42.179967164993286s
pdf_test1(A*):
动作序列如下:
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
步数为: 49
所花时间为: 12.63154673576355s
pdf_test2(A*):
动作序列如下:
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
步数为: 48
所花时间为: 109.41198658943176s
pdf_test3(A*):
动作序列如下:
```

可以看到 pdf 中前两个样例都已经正确给出路径，并且速度还算高效，但是对于后两个例子，由于电脑的内存受限，电脑内存 16GB 加上磁盘缓存了接近 7 个 G，依旧没能得出结果。

3.0 版本:

此版本源代码可以在 code 文件夹中 15puzzle4.py 中找到。

针对上述问题,我向身边其他的同学求助,刚好有同学可以使用他学校的实验室的服务器帮我运行测试,据了解,此服务器内存有 128GB。并且尝试从启发式函数方面进行优化,参考网上的资料得知,可以采用结合线性冲突的曼哈顿距离作为启发式函数,具体代码见上文关键代码展示中启发式函数 3。

通过测试,结果如下:

```
zip_test3(优化后):
动作序列如下:
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
步数为: 48
库花时间为: 0.683518342971861758s
zip_test3(优化后):
动作序列如下:
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
步数为: 48
库花时间为: 0.404588127136238469s
zip_test3(优化后):
动作序列如下:
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
步数为: 48
库花时间为: 0.2668612803320416s
zip_test3(优化后):
动作序列如下:
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
步数为: 48
库花时间为: 0.176801401399977559s
zip_test3(优化后):
动作序列如下:
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
步数为: 48
库花时间为: 0.814894723802211914s
zip_test3(优化后):
动作序列如下:
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
步数为: 48
库花时间为: 2.91809983215332s
pdf_test3(A*):
动作序列如下:
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 14 13 9 5 10 6 8 12 7 10 6 7 11 15
步数为: 49
库花时间为: 2.8124563690802404s
pdf_test3(A*):
动作序列如下:
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
步数为: 48
库花时间为: 16.48476721793611s
pdf_test3(A*):
动作序列如下:
3 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
步数为: 50
库花时间为: 87.89795475806104s
pdf_test3(A*):
动作序列如下:
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
步数为: 62
库花时间为: 234.671786199466s
pdf_test3(IDA*):
动作序列如下:
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
步数为: 49
库花时间为: 246.36888850464045s
pdf_test3(IDA*)
```

可以看到 A*算法对于无论是压缩包还是 pdf 的样例都已经给出正确结果并且速度非常可观,而 IDA*算法也已经解决所有压缩包测试和 pdf 中的前两个样例。在优化启发式函数后 IDA*算法仍无法解决 pdf 中最后两个样例,这让我对 IDA*算法实现逻辑是否正确产生怀疑,经过史哲源学长的指导和对路径检测更深入的了解学习,我意识到有部分代码虽然不会影响 IDA*的逻辑正确性,但是却严重的拖慢了其执行速度,该代码段如下:

```
while len(stack) != 0:
    cur = stack.pop()
    if cur[1] - cur[2] == 0:
        new_path = cur[3]
        flag = True
        end = time.time()
        break
    else:
        position = find_x(cur[0], 0)
        i = position[0]
        j = position[1]
        neighbors = [(i+1,j), (i,j+1), (i-1,j), (i,j-1)]
        for neighbor in neighbors:
            if neighbor[0] < 0 or neighbor[0] > 3 or neighbor[1] < 0 or neighbor[1] > 3:
                continue
            temp = cur[0][neighbor[0]][neighbor[1]]
            m_new = swap_elements(cur[0], pos=(i, j), neighbor)
            if m_new in close:
                continue
            h = calculate_h3(m_new)
            g = cur[2] + 1
            f = g + h
            if f <= limit:
                stack.append((m_new, f, g, cur[3] + (temp,)))
                close.add(m_new)
            else:
                limi = min(limi, f)
                close.remove(cur[0])
        if flag == True:
            break
```

此处引入一个 `close` 来进行路径检测，但是 `remove` 操作会使后续的子节点进行扩展出自己的父节点，这点会使深度优先搜索变得深度趋向无穷。

3.1 版本:

此版本源代码可以在 `code` 文件夹中 `15puzzle5.py` 中找到。

在原来 3.0 代码基础上，舍弃维护 `close` 集合，而是采用下列方法进行检测，使其不要产生逆操作：

```
if len(cur[3]) != 0:
    if temp == cur[3][-1]:
        continue
```

进行测试并得到结果如下：

```
zip_test1(迭代加深A*)
动作序列为
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
步数为: 40
所花时间为:0.06623649597167969s
zip_test2(迭代加深A*)
动作序列为
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
步数为: 40
所花时间为:0.0657205581665039s
zip_test3(迭代加深A*)
动作序列为
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
步数为: 40
所花时间为:0.007605075836181641s
zip_test4(迭代加深A*)
动作序列为
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
步数为: 40
所花时间为:0.24144085345916748s
pdf_test1(IDA*)
动作序列为
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
步数为: 49
所花时间为:7.950641870498657s
pdf_test2(IDA*)
动作序列为
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
步数为: 48
所花时间为:45.19873023033142s
pdf_test3(IDA*)
动作序列为
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
步数为: 56
所花时间为:283.2628664970398s
pdf_test4(IDA*)
动作序列为
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
步数为: 62
所花时间为:1028.1976714134216s
按下任意键继续...
```

可以看到最终 IDA* 也已完成全部样例的测试，尽管时间比 A* 相对较长，但也在可接受范围内，至此实验圆满结束。

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）

最终版本得到对 8 个样例的测试结果如下所示：

借助同学实验室的服务器测试运行后得到：



```
zip_test1(A*):
动作序列如下:
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
步数为: 40
所花时间为: 0.041855812072753906s
zip_test2(A*):
动作序列如下:
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
步数为: 40
所花时间为: 0.05881929397583008s
zip_test3(A*):
动作序列如下:
11 10 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
步数为: 40
所花时间为: 0.0035486221313476562s
zip_test4(A*):
动作序列如下:
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
步数为: 40
所花时间为: 0.04626941680908203s
zip_test1(IDA*):
动作序列为
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
步数为: 40
所花时间为: 0.057084083557128906s
zip_test2(IDA*):
动作序列为
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
步数为: 40
所花时间为: 0.06433725357055664s
zip_test3(IDA*):
动作序列为
11 10 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
步数为: 40
所花时间为: 0.007384538650512695s
zip_test4(IDA*):
动作序列为
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
步数为: 40
所花时间为: 0.23592591285705566s
pdf_test1(A*):
动作序列如下:
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
步数为: 49
所花时间为: 2.909799337387085s
pdf_test2(A*):
动作序列如下:
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
步数为: 48
所花时间为: 16.960822105407715s
pdf_test3(A*):
动作序列如下:
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
步数为: 56
所花时间为: 88.66576409339905s
pdf_test4(A*):
动作序列如下:
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
步数为: 62
所花时间为: 246.48067784309387s
pdf_test1(IDA*):
动作序列为
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
步数为: 49
所花时间为: 7.99440860748291s
pdf_test2(IDA*):
动作序列为
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
步数为: 48
所花时间为: 46.6591362953186s
pdf_test3(IDA*):
动作序列为
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
步数为: 56
所花时间为: 292.0313239097595s
pdf_test4(IDA*):
动作序列为
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
步数为: 62
所花时间为: 1004.6668040752411s
按任意键继续...|
```

后续在自己本地的电脑进行测试也能得出结果, 不过对比上面的结果还是看的出一定的速度差距:

```
zip_test1(A*):
动作序列如下:
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
步数为: 40
所花时间为: 0.04121732711791992s
zip_test2(A*):
动作序列如下:
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
步数为: 40
所花时间为: 0.07089686393737793s
zip_test3(A*):
动作序列如下:
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
步数为: 40
所花时间为: 0.003967761993408203s
zip_test4(A*):
动作序列如下:
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
步数为: 40
所花时间为: 0.0564422607421875s
zip_test1(IDA*):
动作序列为
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
步数为: 40
所花时间为: 0.06806111335754395s
zip_test2(IDA*):
动作序列为
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
步数为: 40
所花时间为: 0.07680845260620117s
zip_test3(IDA*):
动作序列为
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
步数为: 40
所花时间为: 0.008882761001586914s
zip_test4(IDA*):
动作序列为
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
步数为: 40
所花时间为: 0.28093981742858887s
pdf_test1(A*):
动作序列如下:
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
步数为: 49
所花时间为: 3.5340800285339355s
pdf_test2(A*):
动作序列如下:
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
步数为: 48
所花时间为: 20.903421878814697s
pdf_test3(A*):
动作序列如下:
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
步数为: 56
所花时间为: 117.05152678489685s
pdf_test4(A*):
动作序列如下:
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
步数为: 62
所花时间为: 342.8382742404938s
pdf_test1(IDA*):
动作序列为
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
步数为: 49
所花时间为: 10.317883253097534s
pdf_test2(IDA*):
动作序列为
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
步数为: 48
所花时间为: 59.55776929855347s
pdf_test3(IDA*):
动作序列为
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
步数为: 56
所花时间为: 378.0584018230438s
pdf_test4(IDA*):
动作序列为
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
步数为: 62
所花时间为: 1311.5707247257233s
按任意键继续...
```

2. 评测指标展示及分析

关于此类的分析，在上文的 4.创新点&优化中已经详细介绍，故不再赘述。

四、 参考资料

参考资料 1: [15 Puzzle \(4 乘 4 谜题\) IDA*\(DFS 策略与曼哈顿距离启发\) 的 C 语言实现 - SimonC531 - 博客园 \(cnblogs.com\)](#)

参考资料 2: [【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](#)

参考资料 3: [Python 基础\(3\)——元组\(tuple\)的定义与基本操作 元组中只包含一个元素时,需要在元素后面添加逗号,否则会被当作运算符使用-CSDN 博客](#)



五、 总结与收获

本次实验中从开始编写程序到能够得到第一次正确输出的时间可能不到后续优化调整时间还有测试时间的十分之一，本人完全没有想到最后可以优化到如此高效的算法效率。在此次实验过程中也学到了很多，如 **A*** 和 **IDA*** 的具体实现，认识到路经检测和环检测的区别，了解到很多库函数其实看起来只有一行反而花的时间非常久，深入的了解到如何去优化算法，如何去将关键步骤花的时间减少，也切切实实的体会到启发式函数的魅力，选对启发式函数对算法的影响是巨大的，最后非常感谢助教和同学们的帮助。