中山大学计算机学院

人工智能

本科生实验报告

（2022 学年春季学期）

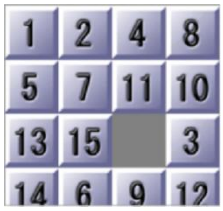
课程名称：Artificial Intelligence

教学班级	信息与计算科学班	专业（方向）	信息与计算科学
学号	21311359	姓名	何凯迪

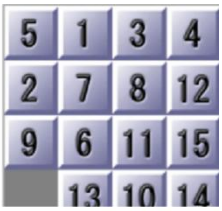
一、 实验题目

使用 A*算法与 IDA*算法解决 15-Puzzle 问题，启发函数可以自己选取，至少尝试两种不同的启发式函数。案例如下：

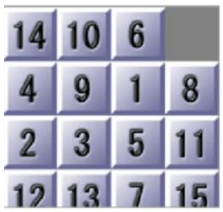
基本样例：



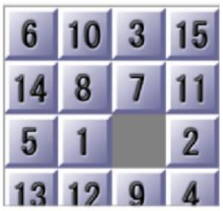
样例1



样例2

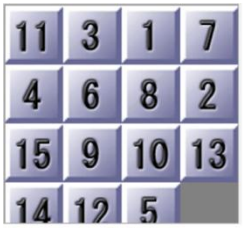


样例3

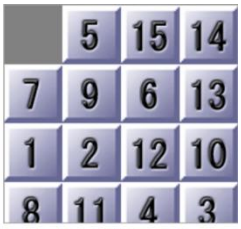


样例4

挑战样例：



样例5



样例6

二、 实验内容

1. 算法原理

A*算法原理：

A*算法是一种启发式搜索算法，用于寻找最优路径。该算法使用两个估价函数来决定每个搜索节点的优先级：一种是从起点到节点的实际成本，另一种是从节点到目标节点的估计成本。A 算法通过最小化这两个成本的和来确定最优路径。

具体步骤如下：

1、初始化一个开放列表（open list）和一个关闭列表（closed list）。起点加入开放列表，并设置起点的估价函数值为 0。

2、从开放列表中选取估价函数值最小的节点作为当前节点。如果当前节点是目标节点，则搜索结束并返回路径。

3、如果当前节点不是目标节点，则将其从开放列表中移动到关闭列表

中，并检查其相邻节点。

4、对于每个相邻节点，如果它们不在关闭列表中，将它们加入开放列表中，并计算它们的估价函数值。

5、如果相邻节点已经在开放列表中，则检查从当前节点到它们的实际成本是否更小。如果是，则更新它们的实际成本和估价函数值，并将它们的父节点设置为当前节点。

6、重复步骤 2-5，直到找到目标节点或者开放列表为空（没有可达节点）。

IDA*算法原理：

IDA 算法的主要思路是在 A*算法的基础上，逐步增加限制的最小 F 值，然后以递归的方式进行深度优先搜索，直到找到目标状态为止。

具体步骤如下：

1、初始化一个阈值为起点到目标节点的估价函数值。

2、从起点开始，以 F 值（实际成本+启发式函数值）为限制条件，进行深度优先搜索，直到达到阈值或者找到目标节点。

3、如果找到目标节点，则返回路径；如果没有找到，则将找到的最小阈值作为下一轮搜索的起始阈值。

4、重复步骤 2-3，直到找到目标节点为止。

2. 伪代码

A*算法：

```
定义目标状态goal_state
定义错位数启发函数wrongmatch(state)
    初始化distance为0
    遍历state中的每一个元素：
        如果state[i][j]不等于goal_state[i][j]，则distance加1
    返回distance
定义移动方向directions
定义A*算法函数Astar(initial_state)
    初始化搜索状态
        计算init_distance = wrongmatch(initial_state)
        将initial_state, [], 0和init_distance打包成元组，插入堆heap中
        将initial_state转换成元组形式，加入集合closed中
    进入搜索循环，直到堆heap为空
        取出距离最小的状态(distance, state, path, cost) = heapq.heappop(heap)
        如果当前状态是目标状态(goal_state)，则返回当前路径path和代价cost
        遍历state中的每一个元素：
                如果state[i][j]等于0，则表示当前空格在(i, j)处
        遍历移动方向directions中的每个方向(dx, dy)
                计算下一步位置(nx, ny) = (x + dx, y + dy)
                如果下一步位置(nx, ny)在拼图的范围内(0 <= nx < 4 and 0 <= ny < 4)
                则生成新的状态new_state，表示将空格从(x, y)移动到(nx, ny)的状态
                计算新状态的错位数启发函数new_distance = wrongmatch(new_state)
                如果新状态new_state没有被访问过
                        计算新状态的代价new_cost = cost + 1
                        生成新的路径new_path = path + [str(new_state[x][y])]
                        将新状态、新路径、新代价、new_distance + new_cost打包成元组，
                                插入堆heap中
                        将新状态转换成元组形式，加入集合closed中
    没有找到解决方案，返回-1和空列表[]
读入初始状态initial_state
调用Astar(initial_state)函数，计算最短路径shortest_path和解决步骤solution_steps
输出原始状态initial_state
输出Lower Bound、A optimal solution和解决步骤
```

IDA*算法：

```
1、初始化起始状态和目标状态。
2、定义曼哈顿距离启发函数，用于计算当前状态到目标状态的估计距离。
3、定义移动方向。

4、实现IDA*搜索算法：
    a. 初始化搜索状态：设置距离上限为起始状态的曼哈顿距离，路径为空，已搜索状态集合只包含起始状态。
    b. 迭代加深搜索：循环执行以下操作，直到找到目标状态或无法找到解决方案：
        i. 执行深度优先搜索：传入当前状态、当前距离、距离上限、路径和已搜索状态集合，并获取搜索结果（下一步的距离估计值、是否找到目标状态、
        ii. 如果找到目标状态，返回路径和距离。
        iii. 如果未找到目标状态，则将距离上限更新为下一步的距离估计值。
    c. 返回-1和空路径（未找到解决方案）。

5、实现深度优先搜索函数：
    a. 如果当前状态到目标状态的距离估计值加上已经走过的距离超过距离上限，返回当前距离加上曼哈顿距离的估计值、未找到目标状态和空路径。
    b. 如果当前状态是目标状态，返回当前距离、找到目标状态和当前路径。
    c. 生成下一步状态：
        i. 获取空格的位置。
        ii. 遍历移动方向，判断是否可以移动，并生成新状态。
        iii. 更新新状态的路径和代价，并继续搜索下一步。
        iv. 如果找到目标状态，返回当前距离、找到目标状态和当前路径。
        v. 如果找到的下一步距离估计值小于距离上限，更新距离上限。
    d. 返回距离上限、未找到目标状态和空路径。

读入初始状态initial_state
调用Astar(initial_state)函数，计算最短路径shortest_path和解决步骤solution_steps
输出原始状态initial_state
输出Lower Bound、A optimal solution和解决步骤
```

3. 关键代码展示（带注释）

本次采用了两种启发函数，一种是曼哈顿距离，一种是错位块数。曼哈顿距离启发函数即计算每块到达目标位置所需移动次数的和；错位块数启发函数即计算所有不在目标位置的块数之和。

两种启发函数实现如下：

曼哈顿启发函数：

```python
# 定义曼哈顿距离启发函数
def manhattan(state):
    distance = 0
    for i in range(4):
        for j in range(4):
            value = state[i][j]
            if value == 0:
                continue
            row, col = (value - 1) // 4, (value - 1) % 4
            distance += abs(row - i) + abs(col - j)
    return distance
```

错位数启发函数：

```python
# 定义错位数启发函数
def manhattan(state):
    distance = 0
    for i in range(4):
        for j in range(4):
            if state[i][j]!=goal_state[i][j]:
                distance += 1
    return distance
```

A*算法：

```
# A*算法
def Astar(initial_state):
    # 初始化搜索状态
    init_distance = manhattan(initial_state)
    heap = [(init_distance, initial_state, [], 0)]
    closed = set()
    closed.add(tuple(map(tuple, initial_state)))

    # 搜索最短路径
    while heap:
        # 取出距离最小的状态
        (distance, state, path, cost) = heapq.heappop(heap)
        if state == goal_state:
            # 找到解决方案，返回路径和代价
            return cost, path

        # 生成下一步状态
        for i in range(4):
            for j in range(4):
                if state[i][j] == 0:
                    # 空格的位置
                    x, y = i, j

        for dx, dy in directions:
            # 移动方向
            nx, ny = x + dx, y + dy
            if 0 <= nx < 4 and 0 <= ny < 4:
                # 可以移动
                new_state = [row[:] for row in state]#列表解析式
                #new_state = []
                #for row in state:
                #    new_row = row[:]
                #    new_state.append(new_row)
                new_state[x][y], new_state[nx][ny] = new_state[nx][ny],new_state[x][y]
                new_distance = manhattan(new_state)

                if tuple(map(tuple, new_state)) not in closed:
                    # 未访问过的状态，加入队列
                    new_path = path+[str(new_state[x][y])]
                    new_cost = cost + 1
                    # 选择保留最小的状态
                    if new_distance <= distance:
```

```
                    heapq.heappush(heap, (new_distance + new_cost, new_state,
new_path, new_cost))

                closed.add(tuple(map(tuple, new_state)))
    # 未找到解决方案
    return -1, []
```

IDA*算法：

```
# IDA*算法
def IDAstar(initial_state):
    # 初始化搜索状态
    limit = manhattan(initial_state)
    path = []
    closed = set()
    closed.add(tuple(map(tuple, initial_state)))

    # 迭代加深搜索
    while True:
        # 执行深度优先搜索
        distance, found , path = dfs(initial_state, 0, limit, path, closed)
        if found:
            # 找到解决方案，返回路径和代价
            return distance, path

        # 增加距离上限
        limit = distance

    # 未找到解决方案
    return -1, []
# 深度优先搜索函数
def dfs(state, distance, limit, path, closed):
    # 判断是否达到距离上限
    if distance + manhattan(state) > limit:
        return distance + manhattan(state), False, []
    if state == goal_state:
        # 找到解决方案，返回路径和代价
        return distance, True, path

    # 生成下一步状态
    for i in range(4):
        for j in range(4):
            if state[i][j] == 0:
                # 空格的位置
                x, y = i, j
```

```
    for dx, dy in directions:
        # 移动方向
        nx, ny = x + dx, y + dy
        if 0 <= nx < 4 and 0 <= ny < 4:
            # 可以移动
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            new_path = path + [str(new_state[x][y])]
            new_cost = distance + 1
            # 继续搜索
            min_distance, found, solution_path = dfs(new_state, new_cost, limit,
new_path, closed)
            if found:
                # 找到解决方案，返回路径和代价
                return min_distance, True, solution_path
            # 更新阈值
            if min_distance < limit:
                limit = min_distance
    # 未找到解决方案
    return limit, False, []
```

4. 创新点&优化（如果有）

```
                new_state = [row[:] for row in state]#列表解析式
                #new_state = []
                #for row in state:
                #    new_row = row[:]
                #    new_state.append(new_row)
```

该处使用列表切片 row[:]而不是 row.copy()或 list(row)，因为切片可以创建一个新列表且与原始列表具有相同的值的，但指向不同的内存地址，这样创建新列表的速度比使用 copy()或 list()函数更快

```
# 此处实现自定义输入
initial_state = []
for i in range(4):
    row = input().split()
    row = [int(num) for num in row]
    initial_state.append(row)
```

除了要求的六个样例外，还实现了自定义输入二维矩阵以解决其它情况。

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）

（样例结果在使用曼哈顿距离启发函数的 A*算法给出，故后续性能对比部分仅用样例 2 做对比）

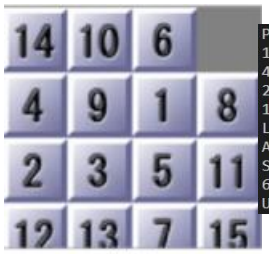
使用 A*算法及曼哈顿距离启发函数结果如下：

样例1

```
PS D:\python文件\exp4> & D:/PY/python.exe d:/python文件/exp4/puzzle_Manhattan.py
1    2    4    8
5    7    11   10
13   15   0    3
14   6    9    12
LowerBound 20 moves
A optimal solution 22 moves
Solution steps:
15 6 9 15 3 10 11 3 10 11 8 4 3 7 6 9 14 13 9 10 11 12
Used time  0.0030012130737304688  sec
```

样例2

```
PS D:\python文件\exp4> & D:/PY/python.exe d:/python文件/exp4/puzzle_Manhattan.py
5    1    3    4
2    7    8    12
9    6    11   15
0    13   10   14
LowerBound 15 moves
A optimal solution 15 moves
Solution steps:
13 10 14 15 12 8 7 2 5 1 2 6 10 14 15
Used time  0.0009999275207519531  sec
```

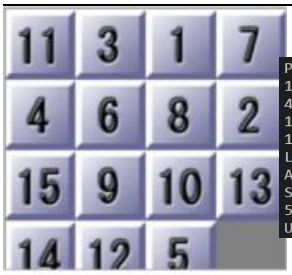


样例3

```
PS D:\python文件\exp4> & D:/PY/python.exe d:/python文件/exp4/puzzle_Manhattan.py
14   10   6    0
4    9    1    8
2    3    5    11
12   13   7    15
LowerBound 35 moves
A optimal solution 53 moves
Solution steps:
6 10 9 4 14 9 10 1 4 10 1 4 10 3 2 14 9 1 3 2 5 11 8 6 4 3 2 5 14 12 13 14 12 9 5 10 6 8 11 12 10 6 8 11 12 7 15 12 11 8 7 11 12
Used time  199.0539412498474  sec
```

样例4

```
PS D:\python文件\exp4> & D:/PY/python.exe d:/python文件/exp4/puzzle_test.py
6    10   3    15
14   8    7    11
5    1    0    2
13   12   9    4
LowerBound 32 moves
A optimal solution 48 moves
Solution 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
Used time  73.32398104667664  sec
```



样例5

```
PS D:\python文件\exp4> & D:/PY/python.exe d:/python文件/exp4/puzzle_Manhattan.py
11   3    1    7
4    6    8    2
15   9    10   13
14   12   5    0
LowerBound 34 moves
A optimal solution 56 moves
Solution 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
Used time  8320.62572813034  sec
```

使用 A*算法及错位块数启发函数结果如下：

样例2

```
PS D:\python文件\exp4> & D:/PY/python.exe d:/python文件/exp4/puzzle_wrongmatch.py
5    1    3    4
2    7    8    12
9    6    11   15
0    13   10   14
LowerBound 12 moves
A optimal solution 15 moves
Solution steps:
13 10 14 15 12 8 7 2 5 1 2 6 10 14 15
Used time  0.0010006427764892578  sec
```

使用 IDA*算法及曼哈顿距离启发函数结果如下：

样例2

```
PS D:\python文件\exp4> & D:/PY/python.exe d:/python文件/exp4/puzzle_IDA_Manhattan.py
5    1    3    4
2    7    8    12
9    6    11   15
0    13   10   14
LowerBound 15 moves
A optimal solution 15 moves
Solution steps:
13 10 14 15 12 8 7 2 5 1 2 6 10 14 15
Used time  1.0004043579101562  ms
```

2. 评测指标展示及分析

启发函数性能对比（在 A*算法下运行样例 2）：

	曼哈顿距离启发函数	错位块数启发函数
运行时间(ms)	0.9999275207519531	1.0006427764892578

如表格所示，在同一算法下，曼哈顿距离启发函数较错位块数启发函数优秀，且实际运行其它案例后发现该结论同样成立。尤其是越复杂的情况，后者的运行时间远远超过前者。

A*及 IDA*算法性能对比：

	优点	缺点
A*算法	能够找到最短路径，是一种最优解算法。 对于更好的启发式函数能够得到更优的性能。	对于搜索空间较大的问题，需要存储大量节点，空间复杂度可能变得较高。
IDA*算法	能克服 A*算法在空间复杂度方面的限制，处理搜索空间较大的问题。 可以在迭代过程中动态调整搜索策略。	每次迭代需要重复搜索大量节点，时间复杂度较高。

估计值和真实值的差距会造成算法性能差距的原因：

1、误导搜索算法：估计值和真实值之间的差距越大，搜索算法在搜索过程中就越容易被误导。例如，如果使用了一个不准确的估价函数，那么搜索算法可能会优先探索到一些不是最优解的状态，从而延长了搜索时间和搜索路径。

2、影响启发式函数的性能：启发式函数是搜索算法中用来评估当前状态与目标状态之间距离的函数。如果估价函数的估计值和真实值之间的差距较大，那么启发式函数的表现也会受到影响，从而导致搜索算法的性能变差。

四、 参考资料

https://blog.csdn.net/weixin_34893782/article/details/117165616?ops_request_misc=&request_id=&biz_id=102&utm_term=15puzzle&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-1-117165616.142^v83^insert_down1,239^v2^insert_chatgpt&spm=1018.2226.3001.4187