课程实验-搜索

- 学号：1813075
- 姓名：刘茵
- 操作系统: Windows 10
  IDE: PyCharm 2020.2.2 x64
  编译语言：Python （v3.7.0)

> 实验报告规范：
>
> 1.实验原理算法说明
> 2.编程环境/编程语言说明
> 3.测试方式说明
> 4.简单实验结果展示

ReadMe：

1. 使用DFS BFS 及A* 算法解决了八数码和十五数码问题
2. 实现了所有算法的结果可视化（为省略内容在A*处具体展示八数码和十五数码的可视化）
3. 代码处注明 #-15：的注释部分为15数码

   注明 #-8： 的注释部分为8数码

测试方式说明：

配置头文件copy | time | matplotlib.pyplot | math | heapq

1. eight_DFS.py

   选择IDE打开文件，运行代码，输入8/15跳转八数码和十五数码，按需输入**搜索深度**，按照**数字+空格+换行**的形式输入初始8/15数码，等待结果和可视化的输出。

2. eight_BFS.py

   选择IDE打开文件，运行代码，输入8/15跳转八数码和十五数码，按照**数字+空格+换行**的形式输入初始8/15数码，等待结果和可视化的输出。

3. A_star.py

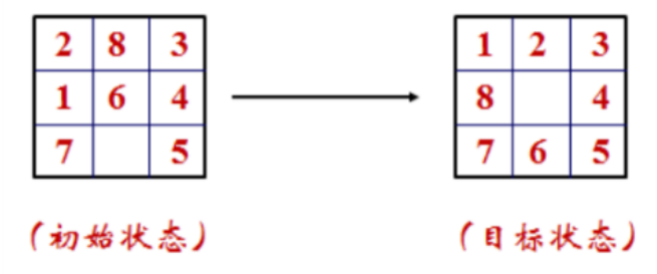
   选择IDE打开文件，运行代码，输入8/15跳转八数码和十五数码，按照**数字+空格+换行**的形式输入初始8/15数码，等待结果和可视化的输出。

4. 在进行算法时间分析时讲可视化函数注释，避免外界干扰。生成的可视化图片可保存到本地。

Ⅰ. 问题介绍

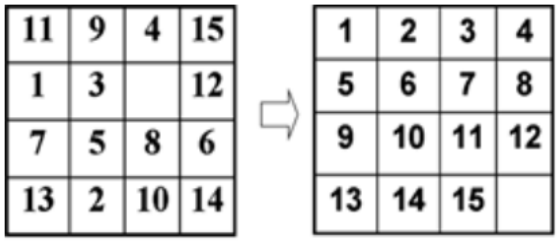
1. 八数码

八数码问题也称为九宫问题。在3×3的棋盘，摆有八个棋子，每个棋子上标有1至8的某一数字，不同棋子上标的数字不相同。棋盘上还有一个空格，与空格相邻的棋子可以移到空格中。要求解决的问题是：给出一个初始状态和一个目标状态，找出一种从初始转变成目标状态的移动棋子步数最少的移动步骤。所谓问题的一个状态就是棋子在棋盘上的一种摆法。棋子移动后，状态就会发生改变。解八数码问题实际上就是找出从初始状态到达目标状态所经过的一系列中间过渡状态。



2. 十五数码

数码问题常被用来演示如何在状态空间中生成动作序列。一个典型的例子是15数码问题，它是由放在一个4×4的16宫格棋盘中的15个数码(1-15)构成，棋盘中的一个单元是空的，它的邻接单元中的数码可以移到该单元中，通过这样不断地移动数码来改变棋盘布局，使棋盘从给定的初始棋局变为目标棋局



Ⅱ. 相关算法

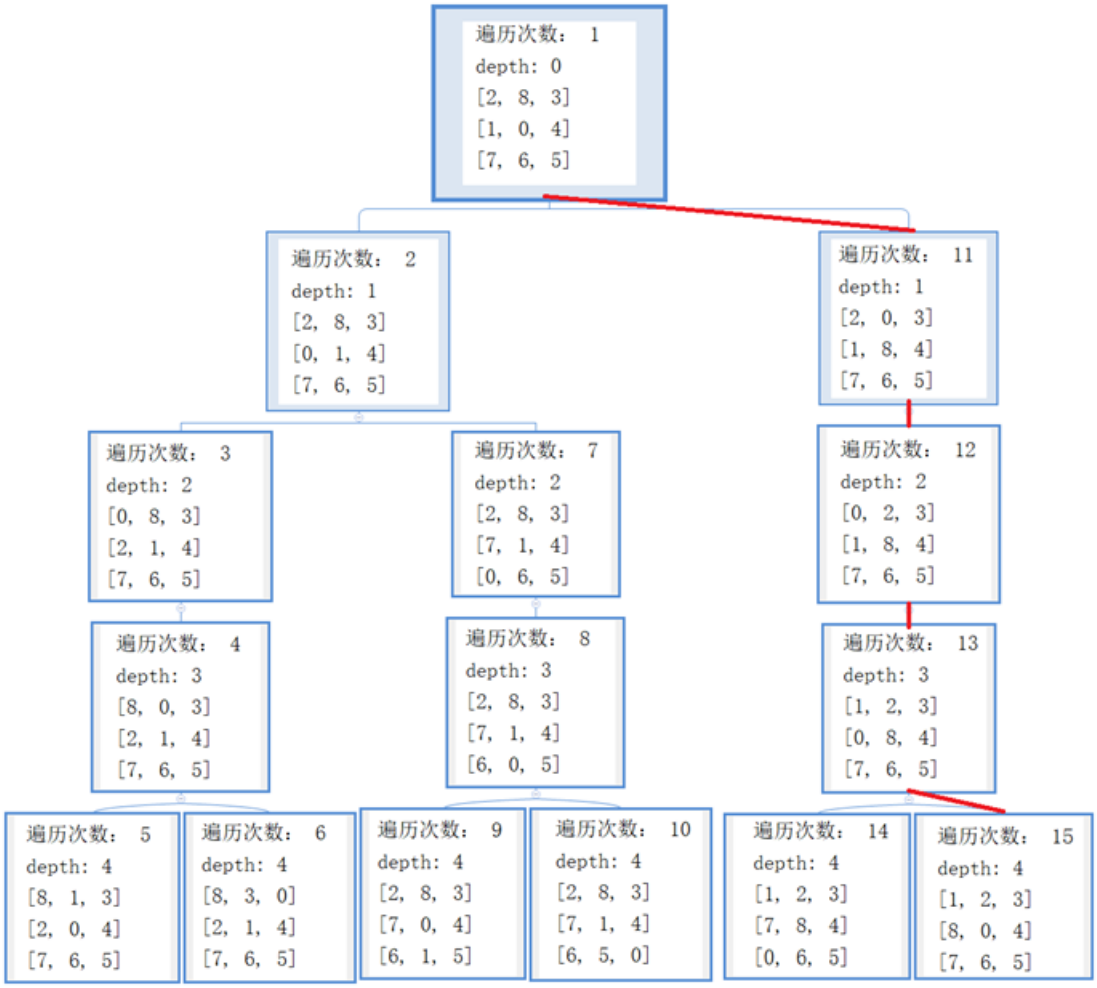
（注：默认为八数码解释）

1. DFS算法

1.1 算法介绍

深度优先搜索算法（Depth-First-Search，DFS）是一种用于遍历或搜索树或图的算法。沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点v的所在边都已被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。属于盲目搜索。

对于八数码问题，没有已经存在的路径供我们遍历，需要我们从初始状态向下延伸（也就是上下左右移动）才能构造出类似的树。



以上图为例。在使用DFS进行搜索时，每个状态都会按照一定的顺序进行上下左右移动（在上图中是左、上、下、右的顺序），一次移动后会产生一个新的状态，然后以新状态为起点继续按约定的顺序（例如先向下）移动。终止的条件是找到解或者达到深度界限。那么如果按照图中下、左、右、上的顺序搜索后的结果将会是最左边的一条路一直是优先向下移动，如果不能向下则依次会是左、右、上的一种。

从遍历次数可以看出DFS是一条道走到黑的找法，一般设置的深度界限是4，所以每一条路最多找到第4层。

1.2 实验代码

代码思路：

（DFS不一定能找到最优解。因为深度界限的原因，找到的解可能在最优解和深度界限之间。）

设置OPEN表，每扩展一个节点，将该节点添加到OPEN表首部（insert(0)，且按照左上下右的**反向**循环）。另外，为了尽可能保证程序有解，在深度优先算法的基础上，添加了深度限制，即在扩展节点之前，计算当前节点的深度，如果深度大于规定的阈值，则不扩展该节点。

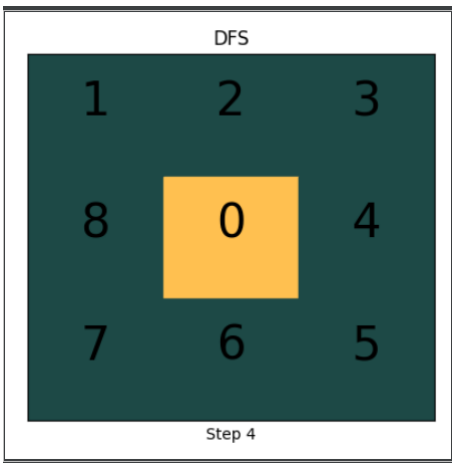
```
def DFS(start, end, generate_child_fn, max_depth):
    """
    DFS 算法
    :参数 start: 起始状态
    :参数 end: 终止状态
    :参数 generate_child_fn: 产生孩子节点的函数
    :参数 max_depth: 最深搜索深度
    :返回: 最优路径长度
    """
    root = State(0, start, hash(str(S0)), None)  # 根节点
    end_state = State(0, end, hash(str(SG)), None)  # 最后的节点
    if root == end_state:
        print("start == end !")

    OPEN.append(root)  # 放入队列

    node_hash_set = set()  # 存储节点的哈希值
    node_hash_set.add(root.hash_value)
    while len(OPEN) != 0:
        global SUM_NODE_flag
        SUM_NODE_flag += 1
        top = OPEN.pop(0)  # 依次出队
        if top == end_state:  # 结束后直接输出路径
            return print_path(top)
        if top.depth >= max_depth:  # 超过深度则回溯
            continue
        # 产生孩子节点，函数纵向延申，孩子节点加入OPEN表
        generate_child_fn(sn_node=top, sg_node=end_state,
hash_set=node_hash_set)
        # def generate_child(sn_node, sg_node, hash_set):
           """
           生成子节点函数
           :参数 sn_node:  当前节点
           :参数 sg_node:  最终状态节点
           :参数 hash_set:  哈希表，用于判重
           :参数 open_table: OPEN表
           :返回: None
           """
    print("在当前深度下没有找到解，请尝试增加搜索深度")  # 没有路径
    return -1
```

1.3 测试结果展示

```
请输入数字：(八数码：8 十五数码：15)
8
搜索深度为：4
请输入初始八数码：
```

```
2 8 3
1 0 4
7 6 5
最终搜索路径为：
------ 0 --------
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
------ 1 --------
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
------ 2 --------
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
------ 3 --------
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
------ 4 --------
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
搜索最优路径长度为 4
搜索时长为 0.000997304916381836 s
共检测节点数为 15
```

- 结果状态（按默认最优路径输出搜索树）



可以看出总共进行了15次遍历，在某一条路的第4层找到了解。

- 其余状态：

```
在当前深度下没有找到解，请尝试增加搜索深度
```

```
所给八数码无解，请检查输入
```

1.4 十五数码DFS

代码：修改目标为4x4棋盘，函数类似于八数码，将父节点的扩展由三行三列变成四行四列。

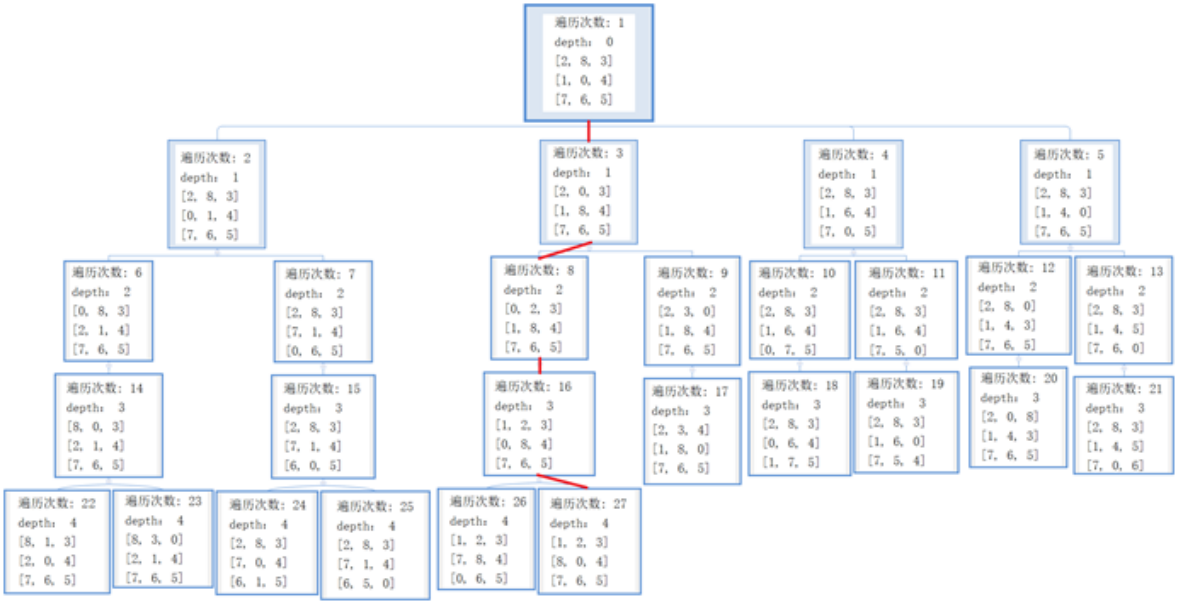
```
请输入数字：(八数码：8 十五数码：15)
15
搜索深度为：5
请输入初始15数码：
1 2 3 4
5 6 7 8
9 10 11 0
13 14 15 12
最终搜索路径为：
------ 0 --------
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 0]
[13, 14, 15, 12]
------ 1 --------
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 0]
搜索最优路径长度为 1
搜索时长为 0.0040242671966552734 s
共检测节点数为 102
```

2. BFS算法

2.1 算法介绍

广度优先搜索算法（Breadth-First-Search，BFS），是一种图形搜索算法。简单的说，BFS是从根节点开始，沿着树的宽度遍历树的节点。如果所有节点均被访问，则算法中止。BFS是一种盲目搜索法，目的是系统地展开并检查图中的所有节点，以找寻结果。

在应用BFS算法进行八数码问题搜索时需要open和closed两个表。首先将初始状态加入open队列，然后进行出队操作并放入closed中，对出队的状态进行扩展（所谓扩展也就是找出其上下左右移动后的状态），将扩展出的状态加入队列，然后继续循环出队-扩展-入队的操作，直到找到解为止



上图这个例子中，BFS横向查找每个节点，依次向下一层搜索，红线的路径是最优搜索顺序。当找到解(target)时一直往前找父节点即可找出求解的移动路线。

2.2 实验代码

通过队列实现：入队-扩展-出队

ps：由于BFS是一层一层找的，所以一定能找到解，并且是最优解。虽然能找到最优解，但它的盲目性依然是一个很大的缺点。从上面的遍历树状图中，每一层都比上一层元素更多，且是近似于指数型的增长。也就是说，深度每增加一，这一层的搜索速度就要增加很多。

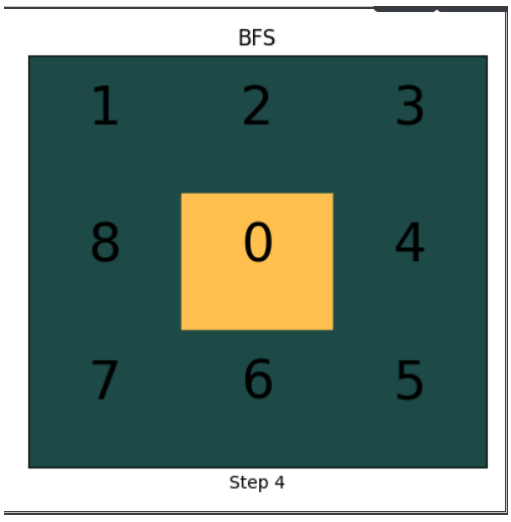
使用哈希数值进行是否达到终点的判断

```
def BFS(start, end, generate_child_fn):
    """
    BFS 算法
    :参数 start: 起始状态
    :参数 end: 终止状态
    :参数 generate_child_fn: 产生孩子节点的函数
    :返回: 最优路径长度
    """
    root = State(0, start, hash(str(S0)), None)  # 根节点
    end_state = State(0, end, hash(str(SG)), None)  # 最后的节点
    if root == end_state:
        print("start == end !")

    OPEN.append(root)  # 放入队列

    node_hash_set = set()  # 存储节点的哈希值
    node_hash_set.add(root.hash_value)
    while len(OPEN) != 0:
        global SUM_NODE_NUM  # 记录BFS搜索的子节点数目
        SUM_NODE_NUM += 1
        top = OPEN.pop(0)  # 依次出队
        if top == end_state:  # 结束后直接输出路径
            return print_path(top)  # 打印最优路径
        # 按照宽度搜索的顺序，依次产生孩子节点，孩子节点加入OPEN队列的末端
        generate_child_fn(sn_node=top, sg_node=end_state,
hash_set=node_hash_set,
                          open_table=OPEN)
        # def generate_child(sn_node, sg_node, hash_set, open_table):
        """
        生成子节点函数
        :参数 sn_node:  当前节点
        :参数 sg_node:  最终状态节点
        :参数 hash_set:  哈希表，用于判重
        :参数 open_table: OPEN表
        :返回: None
        """
    print("无搜索路径!")  # 没有路径
    return -1
```

2.3 实验结果展示

```
请输入初始八数码：
2 8 3
1 0 4
7 6 5
最终搜索路径为：
------ 0 --------
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
------ 1 --------
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
------ 2 --------
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
------ 3 --------
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
------ 4 --------
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
搜索最优路径长度为 4
搜索时长为 3.0054359436035156 s
共检测节点数为 27
```

可知按BFS检测节点的个数为27个，最优路径长度为4.



2.4 十五数码 BFS

```
请输入数字：(八数码：8 十五数码：15)
15
请输入初始15数码：
5 1 2 4
9 6 3 8
13 15 10 11
```

```
0 14 7 12
```
最终搜索路径为：
```
------ 0 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 15, 10, 11]
[0, 14, 7, 12]
------ 1 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 15, 10, 11]
[14, 0, 7, 12]
------ 2 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 0, 10, 11]
[14, 15, 7, 12]
------ 3 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 10, 0, 11]
[14, 15, 7, 12]
------ 4 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 10, 7, 11]
[14, 15, 0, 12]
------ 5 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 10, 7, 11]
[14, 0, 15, 12]
------ 6 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 10, 7, 11]
[0, 14, 15, 12]
------ 7 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[0, 10, 7, 11]
[13, 14, 15, 12]
------ 8 --------
[5, 1, 2, 4]
[0, 6, 3, 8]
[9, 10, 7, 11]
[13, 14, 15, 12]
------ 9 --------
[0, 1, 2, 4]
[5, 6, 3, 8]
[9, 10, 7, 11]
[13, 14, 15, 12]
------ 10 --------
[1, 0, 2, 4]
[5, 6, 3, 8]
[9, 10, 7, 11]
[13, 14, 15, 12]
------ 11 --------
```

```
[1, 2, 0, 4]
[5, 6, 3, 8]
[9, 10, 7, 11]
[13, 14, 15, 12]
------ 12 --------
[1, 2, 3, 4]
[5, 6, 0, 8]
[9, 10, 7, 11]
[13, 14, 15, 12]
------ 13 --------
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 0, 11]
[13, 14, 15, 12]
------ 14 --------
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 0]
[13, 14, 15, 12]
------ 15 --------
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 0]
搜索最优路径长度为 15
搜索时长为 21.332290410995483 s
共检测节点数为 107280
```

3. A*算法

3.1 算法介绍

Astar算法是一种求解最短路径最有效的直接搜索方法，也是许多其他问题的常用启发式算法。

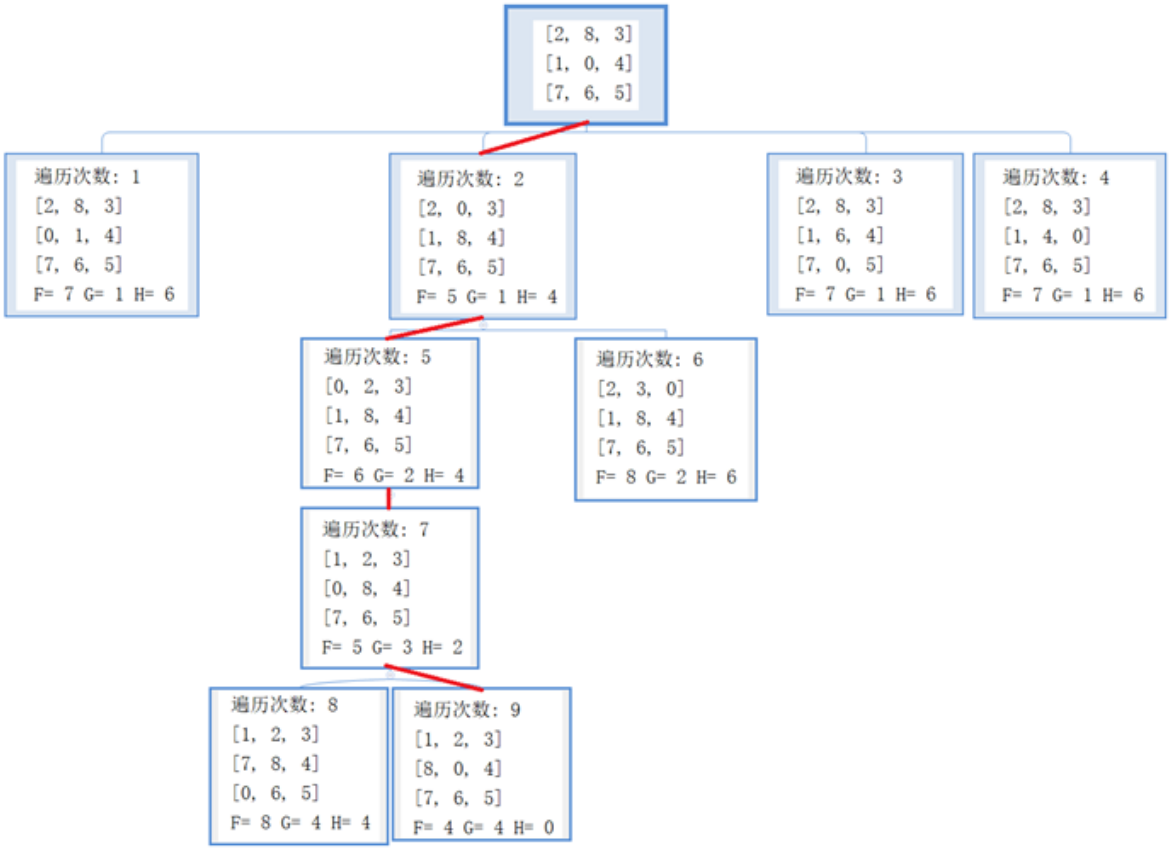
它的启发函数为**f(n)=g(n)+h(n)**,其中，f(n) 是从初始状态经由状态n到目标状态的代价估计，g(n) 是在状态空间中从初始状态到状态n的实际代价，h(n) 是从状态n到目标状态的最佳路径的估计代价。h(n)是启发函数中很重要的一项，它是对当前状态到目标状态的最小代价h*(n)的一种估计，且需要满足 **h(n)<=h'(n)**也就是说h(n)是h'(n)的下界，这一要求保证了Astar算法能够找到最优解。

对于八数码问题：

读入初始状态和目标状态，并计算初始状态评价函数值f；
初始化两个open表和closed表，将初始状态放入open表中
如果open表为空，则查找失败；
否则：
① 在open表中找到评价值最小的节点，作为当前结点，并放入closed表中；
② 判断当前结点状态和目标状态是否一致，若一致，跳出循环；否则跳转到③；
③ 对当前结点，分别按照上、下、左、右方向移动空格位置来扩展新的状态结点，并计算新扩展结点的评价值f并记录其父节点；
④ 对于新扩展的状态结点，进行如下操作：
A．新节点既不在open表中，也不在closed表中，则添加进OPEN表；
B．新节点在open表中，则计算评价函数的值，取最小的。
C．新节点在closed表中，则计算评价函数的值，取最小的。
⑤ 把当前结点从open表中移除；

注：计算距离函数(启发函数h(n))：

1. 采用曼哈顿距离的计算方法，计算每一个位置的数据与它理论位置的横纵标和与纵坐标距离之和。
2. 采用欧氏距离的计算方法，计算每一个位置的数据与它理论位置直线距离之和。



上面输出的解就是按照红色路线标注找到的，从遍历次数和相应状态的启发信息可以看出每次对启发函数值最小的状态进行扩展，依次进行搜索。

3.2 实验代码

A* 算法：
主要在A_star函数中进行，其中，start: 起始状态； end: 终止状态； distance_fn: 距离函数；generate_child_fn: 产生孩子节点的函数；并返回: 最优路径长度。在这其中主体是while循环，只要OPEN表不为空，就一直循环，如果找到目标节点，则直接输出路径，并且返回最优路径长度。
其中，采用heapq堆排序算法，对OPEN表中的节点进行自动排序，每次弹出OPEN表中评价函数值最低的节点，尽可能降低时间复杂度。

```
def A_start(start, end, distance_fn, generate_child_fn):
```

```
    """
    A*算法
    :参数 start: 起始状态
    :参数 end: 终止状态
    :参数 distance_fn: 距离函数，可以使用自定义的
    :参数 generate_child_fn: 产生孩子节点的函数
    :返回: 最优路径长度
    """
    root = State(0, 0, start, hash(str(S0)), None)  # 根节点
    end_state = State(0, 0, end, hash(str(SG)), None)  # 最后的节点
    if root == end_state:
        print("start == end !")

    OPEN.append(root)
    heapq.heapify(OPEN)  # 成堆

    node_hash_set = set()  # 存储节点的哈希值
    node_hash_set.add(root.hash_value)
    while len(OPEN) != 0:
        top = heapq.heappop(OPEN)
        if top == end_state:  # 结束后直接输出路径
            return print_path(top)
        # 产生孩子节点，孩子节点加入OPEN表
        generate_child_fn(sn_node=top, sg_node=end_state,
hash_set=node_hash_set,
                          open_table=OPEN, cal_distence=distance_fn)
    print("无搜索路径!")  # 没有路径
    return -1
```

```python
def cal_E_distence(cur_state):
    """
    计算曼哈顿距离
    :参数 state: 当前状态,4*4的列表, State.state
    :返回: M_cost 每一个节点计算后的曼哈顿距离总和
    """
    E_cost = 0
    for i in range(3):
        for j in range(3):
            # - 15:
            # for i in range(4):
            #     for j in range(4):
            if cur_state[i][j] == SG[i][j]:
                continue
            num = cur_state[i][j]
            if num == 0:
                x, y = 3, 3
            else:
                x = num / 4  # 理论横坐标
                y = num - 4 * x - 1  # 理论的纵坐标
                E_cost += math.sqrt((x - i) * (x - i) + (y - j) * (y - j))
    return E_cost
```

```python
def cal_M_distence(cur_state):
    """
    计算曼哈顿距离
    :参数 state: 当前状态,4*4的列表, State.state
    :返回: M_cost 每一个节点计算后的曼哈顿距离总和
    """
    M_cost = 0
    # -15:
    # for i in range(4):
    #     for j in range(4):
    for i in range(3):
        for j in range(3):
            if cur_state[i][j] == SG[i][j]:
                continue
            num = cur_state[i][j]
            if num == 0:
                x, y = 3, 3
            else:
                x = num / 4  # 理论横坐标
                y = num - 4 * x - 1  # 理论的纵坐标
                M_cost += (abs(x - i) + abs(y - j))
    return M_cost
```

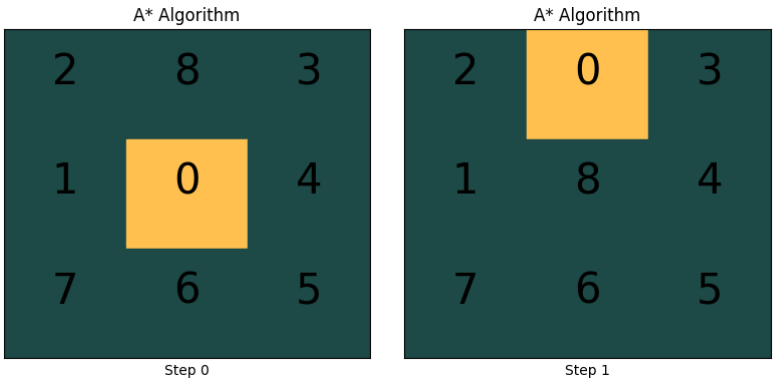
3.3 实验结果展示

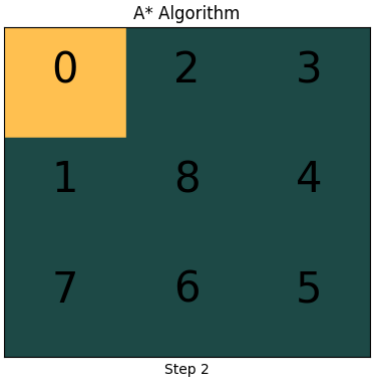
- 欧式距离

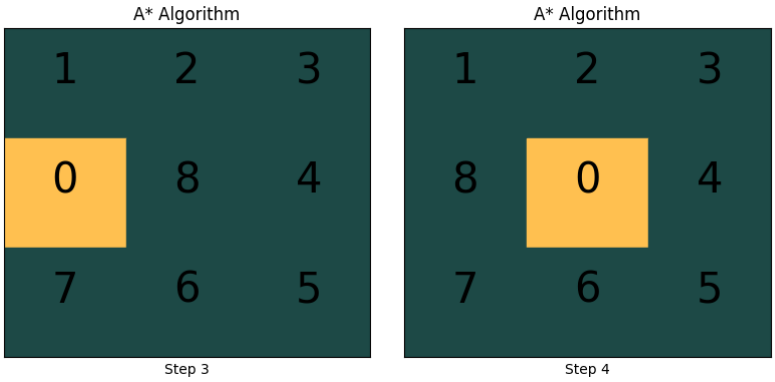
```
请输入初始八数码:
2 8 3
1 0 4
7 6 5
选择距离计算方法(E:欧式距离计算启发函数，M:曼哈顿式距离计算启发函数
E
```

```
最终搜索路径为:
------ 0 --------
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
------ 1 --------
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
------ 2 --------
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
------ 3 --------
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
------ 4 --------
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
采用欧式距离计算启发函数
搜索最优路径长度为 4
搜索时长为 0.0009984970092773438 s
共检测节点数为 9
```

可视化







- 曼哈顿距离

选择距离计算方法(E:欧式距离计算启发函数，M:曼哈顿式距离计算启发函数
M
最终搜索路径为:

```
------ 0 --------
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
------ 1 --------
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
------ 2 --------
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
------ 3 --------
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
------ 4 --------
[1, 2, 3]
```

```
[8, 0, 4]
[7, 6, 5]
采用曼哈顿距离计算启发函数
搜索最优路径长度为 4
搜索时长为 0.0009987354278564453 s
共检测节点数为 9
```

- 其他结果

```
请输入初始八数码:
2 8 3
1 4 0
5 6 7
选择距离计算方法(E:欧式距离计算启发函数，M:曼哈顿式距离计算启发函数
M
无搜索路径!
```

```
请输入初始八数码:
1 2 3
8 0 4
7 6 5
选择距离计算方法(E:欧式距离计算启发函数，M:曼哈顿式距离计算启发函数
E
start == end !
```

3.4 十五数码 A_star+可视化

修改部分细节即可（讲移动越界范围修改，修改可视化图形面积）
此处以欧式距离的结果举例：

```
请输入初始十五数码:
5 1 2 4
9 6 3 8
13 15 10 11
0 14 7 12
选择距离计算方法(E:欧式距离计算启发函数，M:曼哈顿式距离计算启发函数
E
最终搜索路径为:
------ 0 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 15, 10, 11]
[0, 14, 7, 12]
------ 1 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 15, 10, 11]
[14, 0, 7, 12]
------ 2 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 0, 10, 11]
[14, 15, 7, 12]
------ 3 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 10, 0, 11]
[14, 15, 7, 12]
```

```
------ 4 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 10, 7, 11]
[14, 15, 0, 12]
------ 5 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 10, 7, 11]
[14, 0, 15, 12]
------ 6 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[13, 10, 7, 11]
[0, 14, 15, 12]
------ 7 --------
[5, 1, 2, 4]
[9, 6, 3, 8]
[0, 10, 7, 11]
[13, 14, 15, 12]
------ 8 --------
[5, 1, 2, 4]
[0, 6, 3, 8]
[9, 10, 7, 11]
[13, 14, 15, 12]
------ 9 --------
[0, 1, 2, 4]
[5, 6, 3, 8]
[9, 10, 7, 11]
[13, 14, 15, 12]
------ 10 --------
[1, 0, 2, 4]
[5, 6, 3, 8]
[9, 10, 7, 11]
[13, 14, 15, 12]
------ 11 --------
[1, 2, 0, 4]
[5, 6, 3, 8]
[9, 10, 7, 11]
[13, 14, 15, 12]
------ 12 --------
[1, 2, 3, 4]
[5, 6, 0, 8]
[9, 10, 7, 11]
[13, 14, 15, 12]
------ 13 --------
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 0, 11]
[13, 14, 15, 12]
------ 14 --------
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 0]
[13, 14, 15, 12]
------ 15 --------
[1, 2, 3, 4]
[5, 6, 7, 8]
```

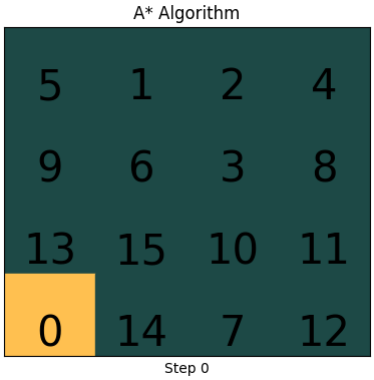
```
[9, 10, 11, 12]
[13, 14, 15, 0]
采用欧式距离计算启发函数
搜索最优路径长度为 15
搜索时长为 0.06084465980529785 s
共检测节点数为 1127
```

可知最优的路径为15步，检测节点数为1127.

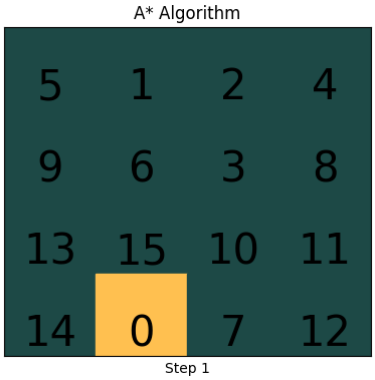
- 可视化：
  - 可视化部分代码
    思路：通过每次打印路径时对图片进行绘制-》进行持续间隔性可视化

```
    def print_path(node):
    """
    输出路径
    :参数 node: 最终的节点
    :返回: None
    """
    print("最终搜索路径为：")
    steps = node.depth

    stack = []  # 模拟栈
    while node.father_node is not None:
        stack.append(node.state)  # 拓展节点
        node = node.father_node
    stack.append(node.state)
    step = 0
    while len(stack) != 0:
        t = stack.pop()  # 先入后出打印
        show_block(t, step)
        # 调用画图，每次打印一个图片，进行可视化
        plot_matrix(t, block=False, plt=plt, zero_color="#FFC050",
another_color="#1D4946",
                    title="A* Algorithm", step=str(step))
        """
        def plot_matrix(matrix, block, plt, zero_color="#93C760",
another_color="blue", title=" ", step=" "):
        plot_matrix: 用来画出矩阵；
        matrix为二维列表；
        plt为画笔，应该为：import matplotlib.pyplot as plt
        """
        step += 1
    return steps  # 返回步数
```

1.



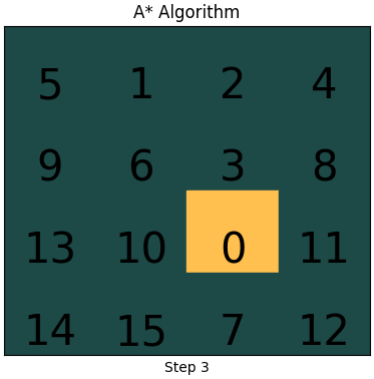
Step 0



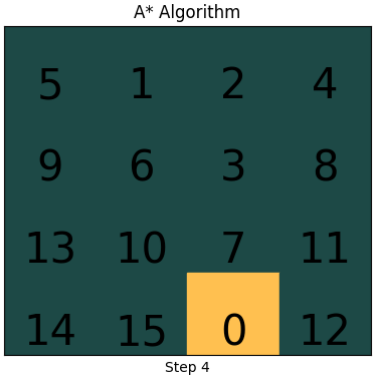
Step 1

Step 2

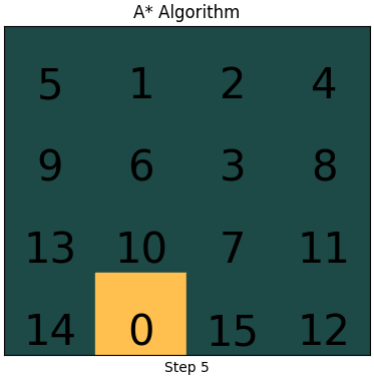
2.



Step 3

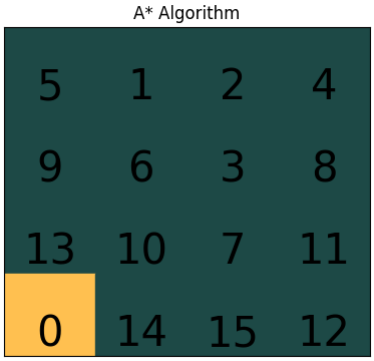


Step 4

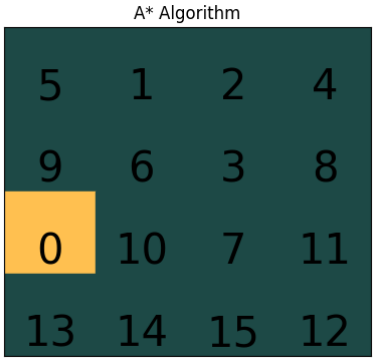


Step 5

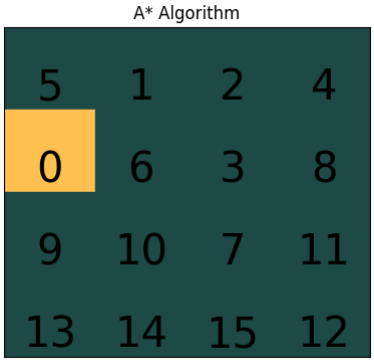
3.



Step 6

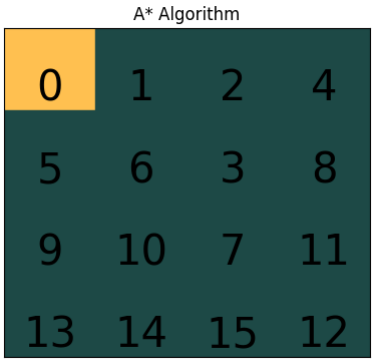


Step 7

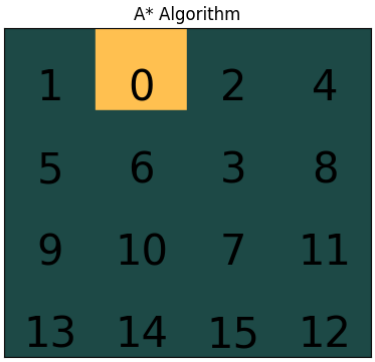


Step 8

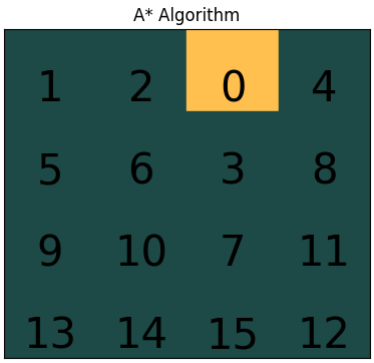
4.



Step 9

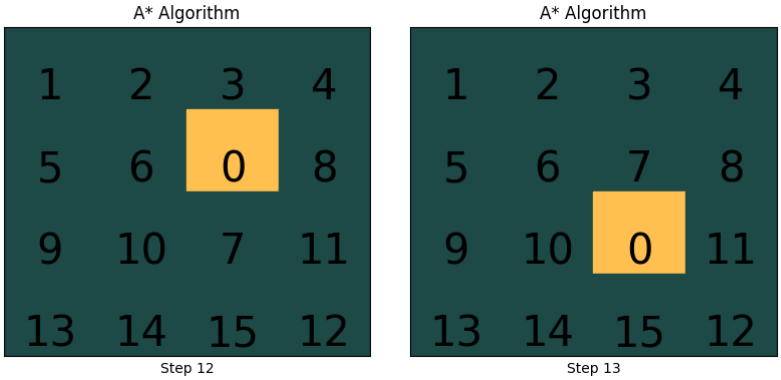


Step 10



Step 11

5.



4. 算法比较

注：15数码为例，记录时间时可视化函数不再调用。

```
情况1：特点：较为简单，容易完成搜索
# S0 = [[5, 1, 2, 4],
#       [9, 6, 3, 8],
#       [13, 15, 10, 11],
#       [0, 14, 7, 12]]

情况2：
# S0 = [[11, 9, 4, 15],
#       [1, 3, 0, 12],
#       [7, 5, 8, 6],
#       [13, 2, 10, 14]]
```

4.1 有界深度优先算法VS广度优先算法VS A*启发式搜索：

	启发函数计算方法	搜索最优路径长度	搜索时长	共检测节点数
情况1	有界深度优先(max_deepth=25)	39	160.8200s	1342460
情况1	广度优先 BFS	15	11.1250s	107285
情况1	A星(曼哈顿距离)	21	0.1253s	2116

注：

```
请输入数字：（八数码：8 十五数码：15）
15
搜索深度为：25
请输入初始15数码：
5 1 2 4
9 6 3 8
13 15 10 11
0 14 7 12
在当前深度下没有找到解，请尝试增加搜索深度
```

通过对比分析，可以看出

A星算法的搜索时长和检测节点数明显小于另外两种方法，可见启发式信息对于搜索过程的重要性；

DFS有界深度优先算法的算法性能差异较大，设置不同的最深深度得到的结果有一定的差异，一般设置较小可能不会在此深度找到结果，较大会造成内存爆炸的现象，所以通过该方法进行搜索较为困难，对于任务较为复杂的情况，很难快速求解。

BFS广度优先算法，针对较为简单问题，基本可以以最短路径给出答案，但同时搜索时间和搜索节点数一定会比启发式搜索多一些，针对复杂问题，很难给出答案，每扩展一层，都会以指数的形式增加待扩展节点的数量，很难得出答案。

综上所述，与深度优先和广度优先算法相比，启发式搜索算法有很强的优越性，一般情况下要尽可能去寻找启发函数，添加到代码中辅助进行算法的训练，尽可能缩短程序运行时间，提高程序效率。

4.2 A*不同的代价函数

	启发函数计算方法	搜索最优路径长度	搜索时长	共检测节点数
情况1	欧氏距离	15	0.0608s	1127
情况1	曼哈顿距离	21	0.1107s	2116
情况2	欧氏距离	43	337.8415s	3431727
情况2	曼哈顿距离	45	18.9553s	292617

PS:每次运行结果会有稍许不同

在路径长度提升之后，曼哈顿距离的启发函数优势更为明显。