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	计科二班	专业（方向）	计算机科学与技术
学号	20337263	姓名	俞泽斌

一、 实验题目

- **使用 A*算法解决 15-Puzzle 问题，启发式函数可以自己选取，最好多尝试几种不同的启发式函数**
- **报告要求**

① **报告中需要包含对算法的原理解释**

② **需要包含性能和结果的对比和分析**

③ **如果使用了多种启发式函数，最好进行对比和分析**

④ **需要在报告中分情况分析估价值和真实值的差距会造成算法性能差距的原因。（参考 PPT P10）**

二、 实验内容

1. 算法原理

A*算法的核心主要是在状态空间中的搜索对每一个搜索的位置进行评估，得到最好的位置，再从这个位置进行搜索直到目标，也就是说，在每一个节点进行判断，找一个 fx=hx+gx 最小的节点，然后到那个节点在进行下一次判断是否为最终状态，如果不是的话再进行 hx 与 fx 的计算，并将 gx++，得到新的 fx，再排序寻找下一个子节点。

2. 需要在报告中分情况分析估价值和真实值的差距会造成算法性能差距的原因。

首先就分成四种情况了，

第一种是当 hn=0 的时候，那么对于每一个节点来说，fn 就直接被 gn 决定，也就是迪杰斯特拉算法的思想，

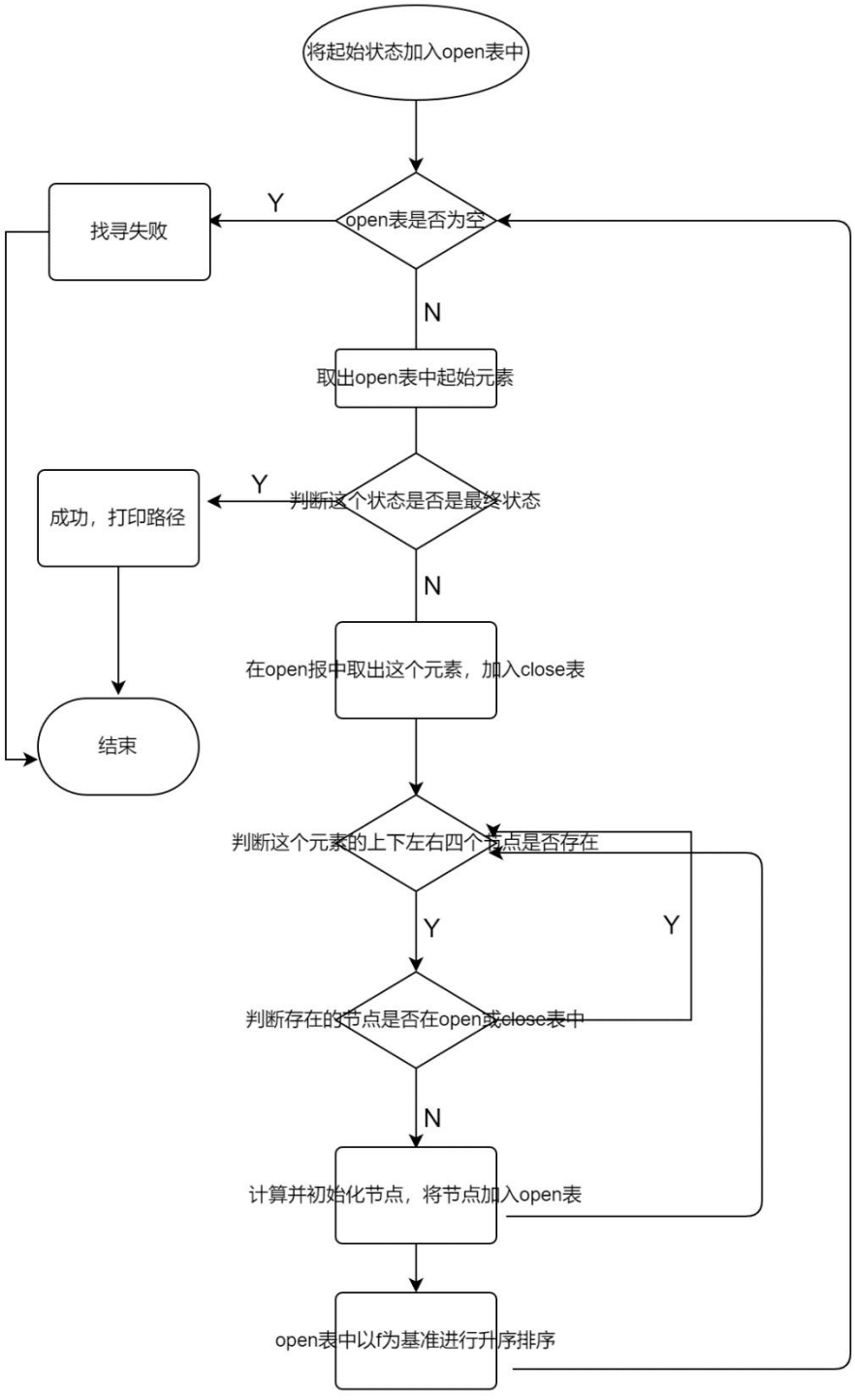
当 hn 比真实的值要小的话，那么我们所找到的节点一直都是最小的那个 fx 的节点，所以只有在排除了 fx 小的节点的时候才可以进入 fx 大的节点上

面来操作，所以肯定可以找到最短的路径。

第三种情况，当 hn 等于真实值的时候，一定是最佳路线进行的。

第四种情况，hn 大于真实值的时候，是可能找到路径的，但是由于在中间的判断上可能选用的不是最好的节点，所以可能找不到最短路径

3. 伪代码



4. 关键代码展示（带注释）

```
class Step:
    state = []
    f = 0
    g = 0
    h = 0
    id = []
    parent = None

    def __init__(self, m):
        self.state = m
        self.f = 0  # f(n)=g(n)+h(n)
        self.g = 0  # g(n)
        self.h = 0  # h(n)
        self.id = []
        self.parent = None
        for i in range(len(m)):
            for j in range(len(m[0])):
                self.id.append(m[i][j])

    def create(self, fn, gn, hn, tparent):
        self.f = fn
        self.g = gn
        self.h = hn
        self.parent = tparent
```

首先是关于类的定义方面，这里定义了一个 step 的类，用来存储每一步中的状态以及 f,g,h 三个函数的值，state 是一个二维列表，id 为一维列表，来存储整个盘面。

```python
# 启发函数
def h(s, finalstep):
    a = 0
    for i in range(len(s.state)):
        for j in range(len(s.state[i])):
            if s.state[i][j] != finalstep.state[i][j]:
                a = a + 1
    return a


def mht(s, finalstep):
    dx = 0
    dy = 0
    tsum = 0
    for i in range(len(s.state)):
        for j in range(len(s.state[i])):
            if s.state[i][j] == 0:
                continue
            dy = abs(j - (s.state[i][j] - 1) % 4)
            dx = abs(i - (s.state[i][j] - 1) // 4)
            tsum = tsum + dx + dy
    return tsum
```

启发式函数，这里主要用了 ppt 上展示的两种，一种 h 就是不在原来位置上的点的个数和，mht 就是曼哈顿距离，也就是每一个点到原来的位置上所需要的步数和，这两个启发式函数用来判断点的优先级

```python
# A*算法
def A_star(s, finalstep):
    tnum = 0
    global openlist
    openlist = [s]
    global closelist
    closelist = set()
    endnum = False
    while len(openlist) > 0 and endnum == False:  # 当open表不为空
        old = openlist[0]  # 取出open表的首节点
        if (old.state == finalstep.state).all():  # 判断是否与目标节点一致
            return old
        openlist.remove(old)  # 将old移出open表
        closelist.add(old)
        # 判断此时状态的空格位置
        for a in range(len(old.state)):
            for b in range(len(old.state[a])):
                if old.state[a][b] == 0:
                    break
            if old.state[a][b] == 0:
                break
```

```
        move = [[1, 0], [-1, 0], [0, 1], [0,-1]]
        for i in range(len(move)):
            na = a + move[i][0]
            nb = b + move[i][1]
            if na >= 4 or na < 0 or nb >= 4 or nb < 0:
                continue
            c = old.state.copy()
            c[a][b] = c[na][nb]
            c[na][nb] = 0
            flag = 1
            new = Step(c)
            tparent = old
            tgn = old.g + 1
            thn = mht(new, finalstep)
            tfn = tgn + thn
            new.create(tfn, tgn, thn, tparent)
            if (new.state == finalstep.state).all():
                endnum = True
                return new
                break
            for k in range(len(openlist)):
                if (new.state == openlist[k].state).all():
                    flag = 0
            if new in closelist:
                flag = 0
            if (flag == 1):
                openlist.append(new)  # 加入open表中
                list_sort(openlist)  # 排序
                print(tnum)
                tnum = tnum + 1
```

接下来是整个 A*的代码方面了，主要的操作其实是定义了两个 open 表和 close 表，首先将初始状态转入 open 表，然后在 open 表不为空的情况下进行循环，开始对 open 表中的首节点操作，判断他是否到了最终状态，如果没有，将这个访问过的节点取出 open 表，放入 close 表，开始判断他有无前后左右的节点，如果有，首先是对这些前后左右的节点是否在 open 和 close 表里的判断，open 表就遍历一遍，close 因为是个 set 就可以直接调用 in 函数，如果都不在，就将这些节点加入 open 表中，并且这些节点都经过了 f,g,h 的初始化，然后进入 open 表之后就可以对整个 open 表进行再一次的排序，以 f 为基准， 进入下一次循环中。

```
def printpath(f):
    tf = []
    tnum = 0
    while f is not None:
        tf.append(f.state)
        f = f.parent
        tnum = tnum + 1
    for i in range(len(tf)):
        print("move", end=" ")
        print(i)
        print(tf.pop())
```

对于找到路径后对路径的打印，这里主要是通过列表的 append 和 pop 操作来实现，因为本来的每一个节点的生成都保留了他的父节点的指针，就通过指针将每一个父节点找出然后加入列表汇总，最后直接打印

```
def main():
    starttime = time.perf_counter()
    sstate = [1, 2, 4, 8, 5, 7, 11, 10, 13, 15, 0, 3, 14, 6, 9, 12]
    fstate = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]
    index = 0
    smatrix = np.zeros((4, 4))
    fmatrix = np.zeros((4, 4))
    for i in range(4):
        for j in range(4):
            smatrix[i][j] = sstate[index]
            fmatrix[i][j] = fstate[index]
            index = index + 1
    startstep = Step(smatrix)  # 初始状态
    finalstep = Step(fmatrix)  # 目标状态

    ans = A_star(startstep, finalstep)
    if ans:
        print("have answer as follow")
        printpath(ans)
    else:
        print("no answer")
    endtime = time.perf_counter()
    print('Running time: %s Seconds' % (endtime - starttime))
```

这就是 main 函数内主要的代码了，开始输入的时候是输入一个一维的数组的，代表地是从上到下从左到右的整个棋盘，然后用 numpy 进行初始化并赋值，然后生成我所需要的类，进行 A*算法的搜索操作，并调用之后进行输出。

5. 创新点&优化（如果有）

```
class Step:
    state = []
    f = 0
    g = 0
    h = 0
    id = []
    parent = None

    def __init__(self, m):
        self.state = m
        self.f = 0  # f(n)=g(n)+h(n)
        self.g = 0  # g(n)
        self.h = 0  # h(n)
        self.id = []
        self.parent = None
        for i in range(len(m)):
            for j in range(len(m[0])):
                self.id.append(m[i][j])

    def create(self, fn, gn, hn, tparent):
        self.f = fn
        self.g = gn
        self.h = hn
        self.parent = tparent
```

类的定义，方便了对于每一个状态进行操作，并且在排序方面可以直接采用 list_sort

```
def list_sort(l):
    cmp = operator.attrgetter('f')
    l.sort(key=cmp)
```

引入了 operator 这个库里的 sttrgetter 函数，来用 f 作为基准来对 open 表进行排序，方便了排序操作

```
closelist = set()
```

```
if new in closelist:
    flag = 0
```

使用 set 来简化对 close 表的遍历，调用 in 函数

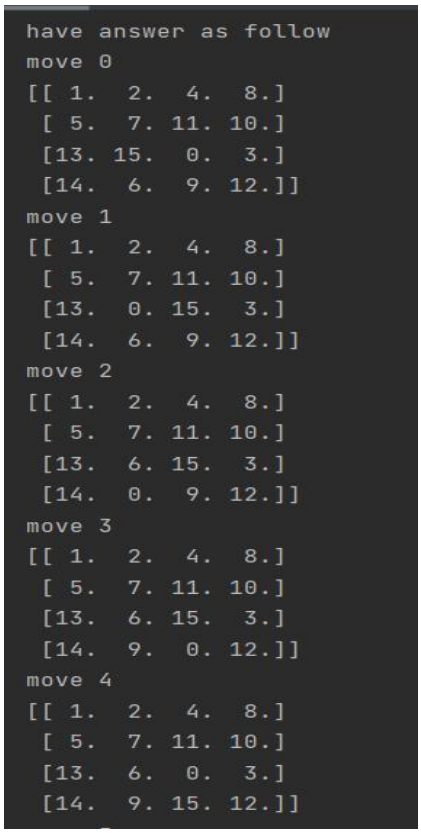
```
endtime = time.perf_counter()
print('Running time: %s Seconds' % (endtime - starttime))
```

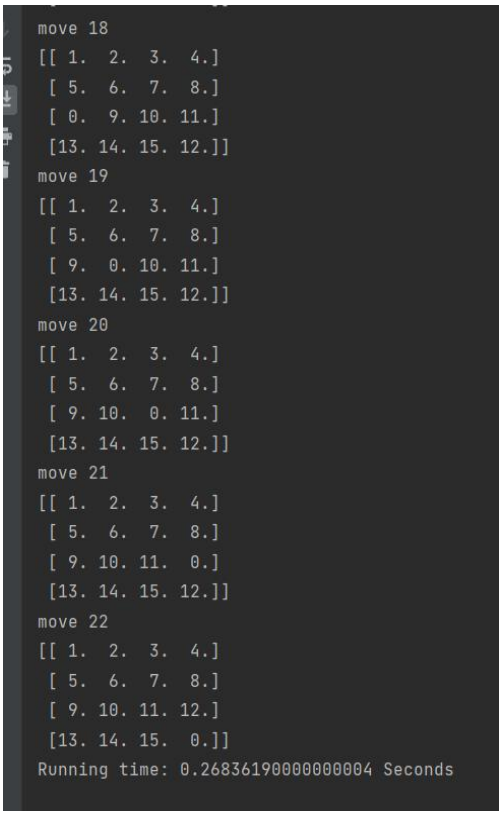
调用 time 模块里的 perf_counter 来记录所耗费的时间，方便比较函数性能

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）

对于样例 1，因为篇幅的限制，我这里只截取开头的和后面的步骤，可以看到样例的循环操作包括代码都是正常实现的，然后运行的时间是 0.268 秒，可以说是比较快速的





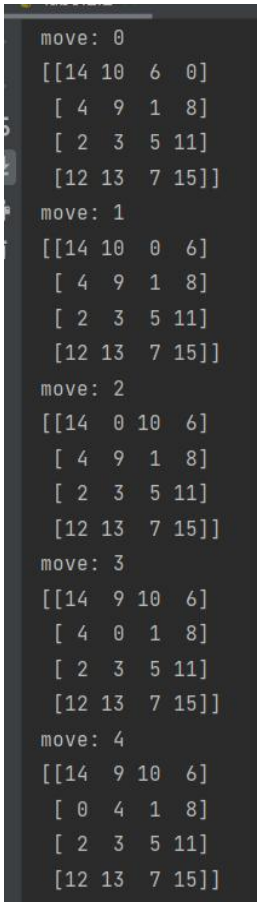
```
move 0
[[ 5.  1.  3.  4.]
 [ 2.  7.  8. 12.]
 [ 9.  6. 11. 15.]
 [ 0. 13. 10. 14.]]
move 1
[[ 5.  1.  3.  4.]
 [ 2.  7.  8. 12.]
 [ 9.  6. 11. 15.]
 [13.  0. 10. 14.]]
move 2
[[ 5.  1.  3.  4.]
 [ 2.  7.  8. 12.]
 [ 9.  6. 11. 15.]
 [13. 10.  0. 14.]]
move 3
[[ 5.  1.  3.  4.]
 [ 2.  7.  8. 12.]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

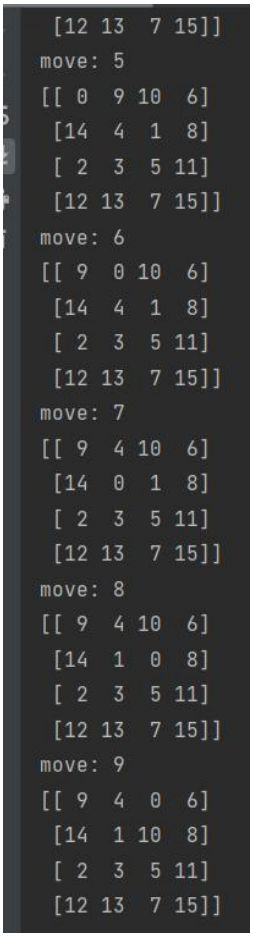
```
move 11
[[ 1.  2.  3.  4.]
 [ 5.  0.  7.  8.]
 [ 9.  6. 11. 12.]
 [13. 10. 14. 15.]]
move 12
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9.  0. 11. 12.]
 [13. 10. 14. 15.]]
move 13
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]
 [13.  0. 14. 15.]]
move 14
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]
 [13. 14.  0. 15.]]
move 15
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]
 [13. 14. 15.  0.]]
Running time: 0.008435499999999985 Seconds
```

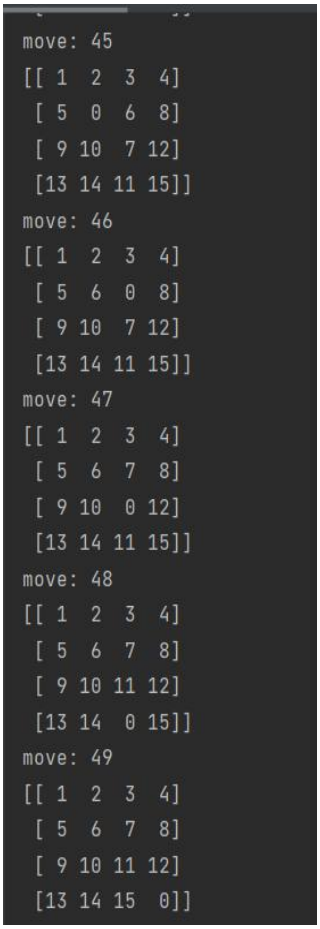
和样例 1 中的一样，样例二也是截取了其中的一部分数据，因为这样例所需步骤比较少，所以可以看到是比较快速的就完成了

样例 3 就跑的非常的慢，因为需要 49 步的原因，而且每一步都需要不断地尝试，所以

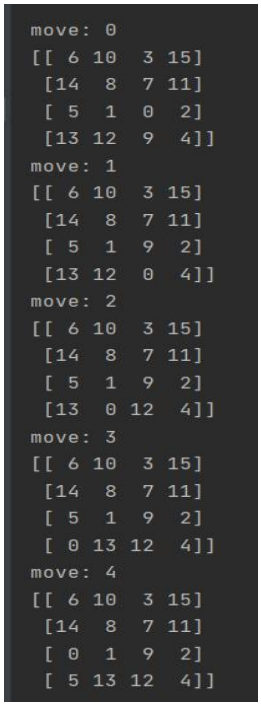
运行时间非常长，但是还是可以跑出最优解来的，也截取部分数据如下

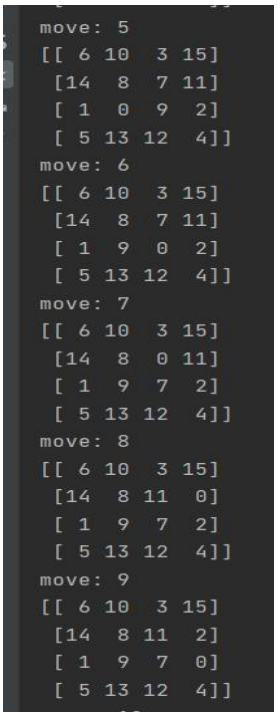


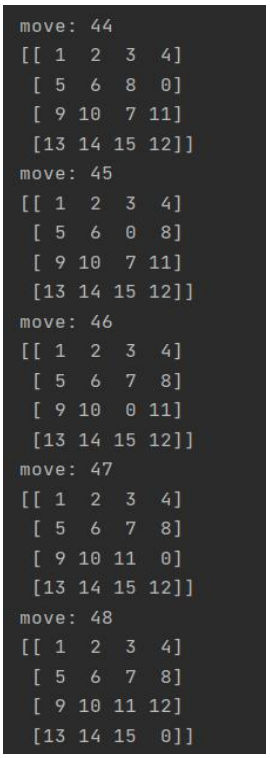




样例 4 也跑的比较慢，因为也需要 48 步的操作，所以单纯的 A*确实非常的慢，但比 3 稍微好一点，毕竟后面加一步就是很大的运算量







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）

对于时间的分析，那主要还是在启发式函数的使用方面，如果使用单纯的不在正确位置上的点的个数作为启发式函数的话，对于样例二这种步骤少的不太明显，但对于样例一就十分明显了，如下图

```
move 19
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9.  0. 10. 11.]
 [13. 14. 15. 12.]]
move 20
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10.  0. 11.]
 [13. 14. 15. 12.]]
move 21
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11.  0.]
 [13. 14. 15. 12.]]
move 22
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]
 [13. 14. 15.  0.]]
Running time: 804.7434106 Seconds
```

这就是采用了不在正确位置上的点的个数作为启发式函数所得到的结果，对比下面的使用曼哈顿距离所得到的时间来说，差距比较大，所以一个好的启发式函数在 A*算法中至关重要

```
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]
 [13.  0. 14. 15.]]
move 14
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]
 [13. 14.  0. 15.]]
move 15
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]
 [13. 14. 15.  0.]]
Running time: 0.063861 Seconds
```

```
move 19
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9.  0. 10. 11.]
 [13. 14. 15. 12.]]
move 20
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10.  0. 11.]
 [13. 14. 15. 12.]]
move 21
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11.  0.]
 [13. 14. 15. 12.]]
move 22
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]
 [13. 14. 15.  0.]]
Running time: 0.26836190000000004 Seconds
```

这是使用了不在正确位置上的点的个数作为启发式函数所得到的样例二的结果，比较下面的使用曼哈顿距离的时间来说不是向前面那么明显，但还是差距比较大的

```
move 13
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]
 [13.  0. 14. 15.]]
move 14
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]
 [13. 14.  0. 15.]]
move 15
[[ 1.  2.  3.  4.]
 [ 5.  6.  7.  8.]
 [ 9. 10. 11. 12.]
 [13. 14. 15.  0.]]
Running time: 0.008881699999999992 Seconds
```

四、 参考资料

https://blog.csdn.net/dujuancao11/article/details/109749219?ops_request_misc=%257B%2522request%255Fid%2522%253A%25221649316816816780357254651%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=164931616816780357254651&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-109749219.142^v6^control,157^v4^control&utm_term=A*%E7%AE%97%E6%B3%95&spm=1018.2226.3001.4187

https://blog.csdn.net/StarInShadow/article/details/102808489?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216493168131678224647146 0%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=164931681316782246471460&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-102808489.142^v6^control,157^v4^control&utm_term=15puzzleA*&spm=1018.2226.3001.4187

https://blog.csdn.net/u011412840/article/details/90739592?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216493168131678224647146 0%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=164931681316782246471460&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-2-90739592.142^v6^control,157^v4^control&utm_term=15puzzleA*&spm=1018.2226.3001.4187

PS：可以自己设计报告模板，但是内容必须包括上述的几个部分，不需要写实验感想