中山大学数据科学与计算机学院

计算机科学与技术专业-人工智能

本科生实验报告

（2018-2019学年秋季学期）

课程名称：**Artificial Intelligence**

|  |  |  |  |
| --- | --- | --- | --- |
| 教学班级 | 计科2班 | 专业（方向） | 计算机科学与技术 |
| 学号 | 16337341 | 姓名 | 朱志儒 |

## 实验题目

**启发式搜索**

## 实验内容

* **算法原理**

**启发式搜索**

启发式搜索又叫有信息的搜索，利用问题所拥有的启发信息来引导搜索，达到减少搜索范围，降低问题复杂度的目的。无信息搜索对所有的可能路径节点一视同仁，启发式搜索可指导搜索向最有希望的方向前进。

我们使用估价函数来估计节点的重要性，其中g(x)是从初始节点到节点x付出的实际代价，h(x)是从节点x到目标节点的最优路径的估计代价。h(x)代表启发式搜索问题中的启发信息，是算法的关键，合理的定义h(x)可让整个搜索算法找到问题的一个最优解。

**A\* 搜索算法**

A\* 搜索算法在BFS算法的基础上加入了启发式信息。算法步骤如下：

1. 从起始节点开始，将其作为待处理的节点加入“开启列表”；
2. 从“开启列表”中选取估价函数值最小的节点C，将其加入“关闭列表”并从“开启列表”中删除；
3. 检查节点C的所有相邻节点，如果节点在“关闭列表”中，则检查另一相邻节点；如果节点不在“开启列表”中，则计算其g(x)和h(x)，设置其父节点为C，将其加入“开启列表”；如果节点在“开启列表”中，则更新其g(x)；
4. 检查“开启列表”是否为空，若为空，则返回NOT\_FOUND；若不为空，则继续下一步；
5. 判断“开启列表”中是否存在目标节点，若存在，则返回FOUND；若不存在，则重复步骤2。

**IDA\* 搜索算法**

IDA\* 搜索算法是迭代加深搜索算法（IDS）的一个扩展，由于不用维护列表，所有其空间复杂度远小于A\* 搜索算法，算法步骤如下：

1. 将阈值设为初始节点的h(x)值；
2. 从起始节点开始进行深度受限搜索
3. 如果没有找到目标节点，则将阈值设为步骤2中返回的值，再重复步骤2；如果找到目标节点，则结束搜索。

深度受限搜索算法步骤：

1. 判断节点的f(x)值是否大于阈值，若大于，则返回该f(x)值，若不大于，则进入下一步；
2. 判断节点是否为目标节点，若是，则返回FOUND，若不是，则进入下一步；
3. 检查节点的所有相邻节点，计算它们的g(x)和h(x)，对于每个相邻节点重复步骤1。

* **伪代码**

**A\* 搜索算法**

function A\_Star(start, end)

closeset := {}

openset := {start}

father\_node := 空的map

g\_value := 默认值为无穷的map

g\_value[start] := 0

f\_value := 默认值为无穷的map

f\_value[start] := heuristic\_cost\_estimate(start, end)

while openset is not empty

current\_node := openset中f\_value值最小的节点

if current\_node = end

return reconstruct\_path(father\_node, current\_node)

openset.remove(current\_node)

closeset.add(current\_node)

for each neighbor\_node of current\_node

if neighbor\_node in closeset

continue

tentative\_g\_value := g\_value[current\_node] + dist\_between(current\_node, neighbor\_node)

if neighbor\_node not in openset

openset.add(neighbor\_node)

else if tentative\_g\_value >= g\_value[neighbor\_node]

continue

father\_node[neighbor\_node] := current\_node

g\_value[neighbor\_node] := tentative\_g\_value

f\_value[neighbor\_node] := g\_value[neighbor\_node] + heuristic\_cost\_estimate(neighbor\_node, end)

**IDA\* 搜索算法**

procedure ida\_star(root)

bound := h\_value[root]

path := [root]

loop

t := search(path, 0, bound)

if t = FOUND

return (path, bound)

if t = infinity

return NOT\_FOUND

bound := t

end loop

end procedure

function search(path, g\_value, bound)

node := path.last

f\_value := g\_value + h\_value(node)

if f\_value > bound

return f\_value

if is\_end(node)

return FOUND

min := infinity

for succ in successors(node)

if succ not in path

path.push(succ)

t := search(path, g\_value + cost(node, succ), bound)

if t = FOUND

return FOUND

if t < min

min := t

path.pop()

return min

* **关键代码**

1. **节点的定义**

class Node:

def \_\_init\_\_(self, matrix, size):

self.matrix = matrix

self.size = size

self.father = None

self.g = 0

self.h = 0

self.change\_number = None

def set\_new\_h(self, final\_matrix):

# 矩阵中每个点距目标位置的曼哈顿距离的和作为H值

for x in range(self.size):

for y in range(self.size):

for i in range(self.size):

for j in range(self.size):

if self.matrix[x][y] == final\_matrix[i][j]:

self.h += abs(x - i) + abs(y - j)

1. **A\* 搜索算法**

找出“开放列表”中F值最小的节点：

def find\_min\_f\_node(self):

# 找出“开放列表”中F值最小的节点

min\_node = self.openlist[0]

for item in self.openlist:

if item.g + item.h < min\_node.g + min\_node.h:

min\_node = item

return min\_node

判断节点是否在list中：

def node\_in\_list(self, node, list):

# 判断节点是否在list中

for item in list:

if item.matrix == node.matrix:

return True

return False

获得节点node在“开放列表”中的下标

def get\_index\_in\_openlist(self, node):

# 获得节点node在“开放列表”中的下标

for i in range(len(self.openlist)):

if node.matrix == self.openlist[i].matrix:

return i

return None

检测一个相邻节点：

def search\_node(self, node):

# 检测相邻节点node

if self.node\_in\_list(node, self.closelist):

# 节点node在“关闭列表”中，直接返回

return

if not self.node\_in\_list(node, self.openlist):

# 节点node不在“开启列表”中，计算其G和H，设置其父节点为当前节点

node.g = self.step

node.set\_new\_h(self.final\_matrix)

node.father = self.current\_node

self.openlist.append(node)

else:

# 节点node在“开启列表”中，更新其g(x)

index = self.get\_index\_in\_openlist(node)

if self.current\_node.g + self.step < self.openlist[index].g:

self.openlist[index].g = self.current\_node.g + self.step

self.openlist[index].father = self.current\_node

搜索路径：

def search(self):

self.start\_node.set\_new\_h(self.final\_matrix)

self.start\_node.g = self.step

self.openlist.append(self.start\_node)

while True:

# 从“开启列表”中选取估价函数值最小的节点，将其加入“关闭列表”并从“开启列表”中删除

self.current\_node = self.find\_min\_f\_node()

self.closelist.append(self.current\_node)

self.openlist.remove(self.current\_node)

self.step = self.current\_node.g

current\_matrix = self.current\_node.matrix

x, y = find\_zero(current\_matrix, self.size)

self.step += 1

# 检查节点C的所有相邻节点

if x + 1 < self.size:

next\_node = Node(move\_zero(deepcopy(current\_matrix), x, y, x + 1, y), self.size)

# 检测相邻节点

self.search\_node(next\_node)

next\_node.change\_number = current\_matrix[x + 1][y]

if x - 1 >= 0:

next\_node = Node(move\_zero(deepcopy(current\_matrix), x, y, x - 1, y), self.size)

# 检测相邻节点

self.search\_node(next\_node)

next\_node.change\_number = current\_matrix[x - 1][y]

if y + 1 < self.size:

next\_node = Node(move\_zero(deepcopy(current\_matrix), x, y, x, y + 1), self.size)

# 检测相邻节点

self.search\_node(next\_node)

next\_node.change\_number = current\_matrix[x][y + 1]

if y - 1 >= 0:

next\_node = Node(move\_zero(deepcopy(current\_matrix), x, y, x, y - 1), self.size)

# 检测相邻节点

self.search\_node(next\_node)

next\_node.change\_number = current\_matrix[x][y - 1]

# 判断“开启列表”中是否存在目标节点

if self.node\_in\_list(Node(self.final\_matrix, self.size), self.openlist):

# 若存在，结束搜索，返回路径

index = self.get\_index\_in\_openlist(Node(self.final\_matrix, self.size))

tmp = self.openlist[index]

path = [tmp]

while tmp.father != None:

tmp = tmp.father

path.append(tmp)

path.reverse()

del path[0]

return path

elif len(self.openlist) == 0:

# 若“开启列表”为空，结束搜索

return

1. **IDA\* 搜索算法**

更新新节点的G和H值：

def search\_node(self, matrix, depth):

# 更新新节点的G和H值

node = Node(matrix, self.size)

node.g = depth

node.set\_new\_h(self.finalnode.matrix)

return node

深度受限搜索：

def subsearch(self, node, pre, depth, limit):

if node.g + node.h > limit:

# 节点的F值大于阈值，结束搜索，将该F值加入cutoff列表

self.cutoff.append(node.g + node.h)

return

if node.matrix == self.finalnode.matrix:

# 节点是目标节点，结束搜索，保存父节点

node.father = pre

self.flag = True

self.endnode = node

return

# 记录父节点

node.father = pre

# 将节点加入已访问的列表

self.visited.append(node.matrix)

x, y = find\_zero(node.matrix, self.size)

# 递归搜索所有相邻节点

if x + 1 < self.size and not self.flag:

next\_matrix = move\_zero(deepcopy(node.matrix),x, y, x + 1, y)

if next\_matrix not in self.visited:

# 节点不在已访问的列表中，更新新节点的G和H值

next\_node = self.search\_node(next\_matrix, depth)

next\_node.change\_number = next\_matrix[x][y]

self.subsearch(next\_node, node, depth + 1, limit)

if x - 1 >= 0 and not self.flag:

next\_matrix = move\_zero(deepcopy(node.matrix),x, y, x - 1, y)

if next\_matrix not in self.visited:

# 节点不在已访问的列表中，更新新节点的G和H值

next\_node = self.search\_node(next\_matrix, depth)

next\_node.change\_number = next\_matrix[x][y]

self.subsearch(next\_node, node, depth + 1, limit)

if y + 1 < self.size and not self.flag:

next\_matrix = move\_zero(deepcopy(node.matrix),x, y, x, y + 1)

if next\_matrix not in self.visited:

# 节点不在已访问的列表中，更新新节点的G和H值

next\_node = self.search\_node(next\_matrix, depth)

next\_node.change\_number = next\_matrix[x][y]

self.subsearch(next\_node, node, depth + 1, limit)

if y - 1 >= 0 and not self.flag:

next\_matrix = move\_zero(deepcopy(node.matrix),x, y, x, y - 1)

if next\_matrix not in self.visited:

# 节点不在已访问的列表中，更新新节点的G和H值

next\_node = self.search\_node(next\_matrix, depth)

next\_node.change\_number = next\_matrix[x][y]

self.subsearch(next\_node, node, depth + 1, limit)

# 回溯时将节点从已访问列表中删除

self.visited.remove(node.matrix)

搜索路径：

def search(self):

# 阈值设为初始节点的H值

limit = self.startnode.h

while not self.flag:

# 没有找到目标节点，将阈值设为cutoff中最小值

self.subsearch(self.startnode, None, 1, limit)

limit = min(self.cutoff)

self.cutoff = []

tmp = self.endnode

# 找到目标节点，返回路径

path = [tmp]

while tmp.father != None:

tmp = tmp.father

path.append(tmp)

path.reverse()

del path[0]

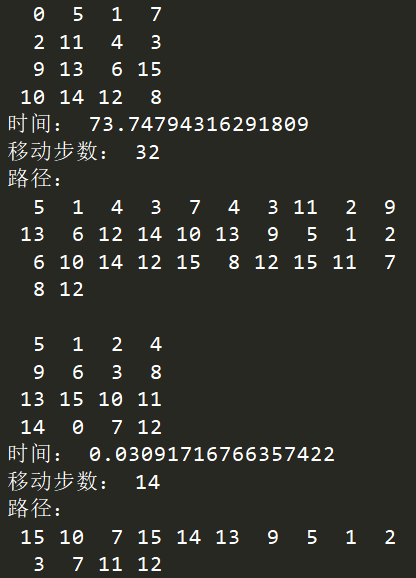
return path

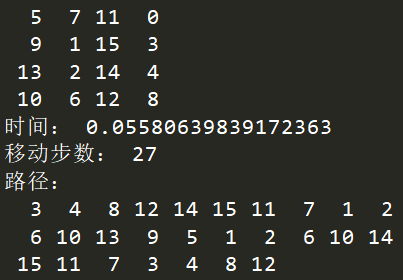
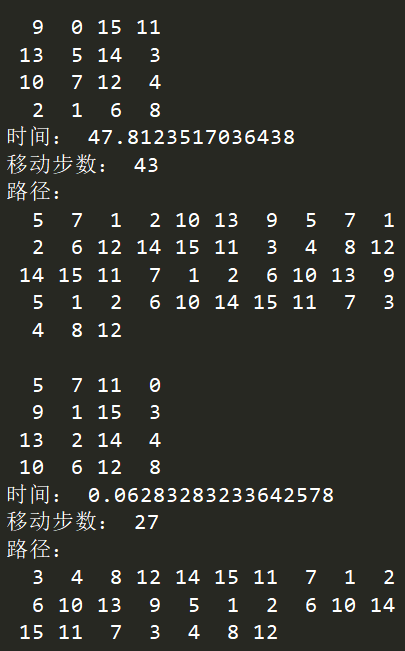
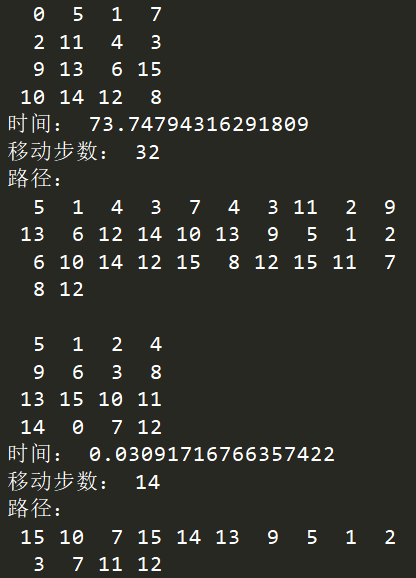
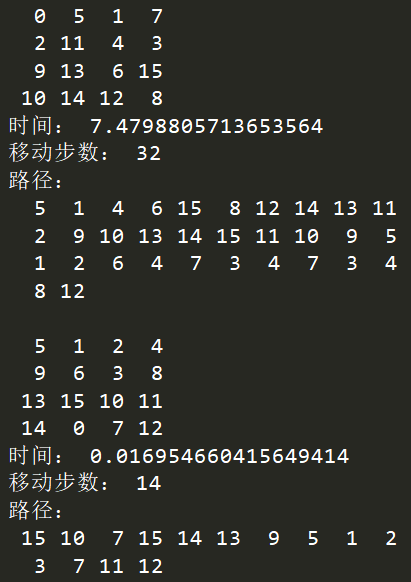
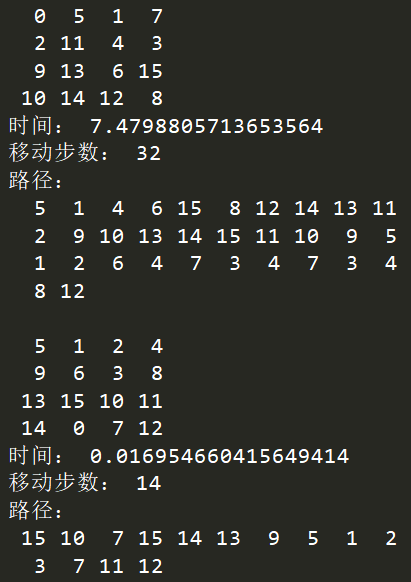
## 实验结果及分析

* **实验结果展示**

**曼哈顿距离：**

IDA\* 算法 A\* 算法

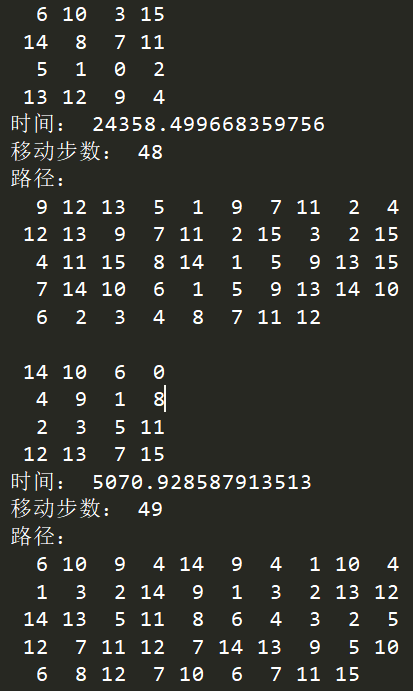
****

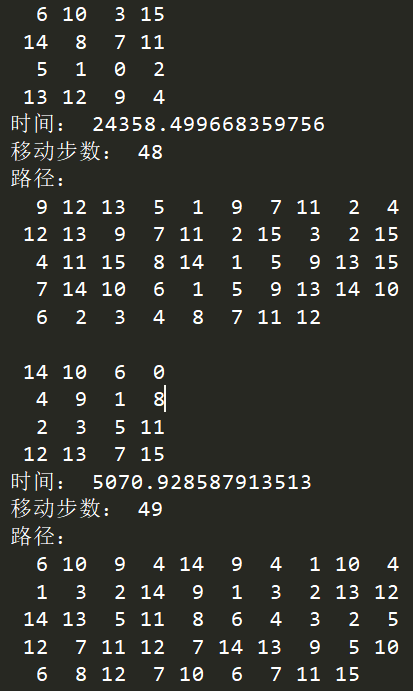


从图中可以看出IDA\* 算法和A\* 算法均可以找到最优路径，当移动步数较小时，两种算法运行时间差距不大，但当移动步数较大时，IDA\* 算法比A\* 算法用时明显较少。

下图是PPT中样例IDA\* 算法运行的结果，而A\* 算法并不能在相同的时间内找到路径。

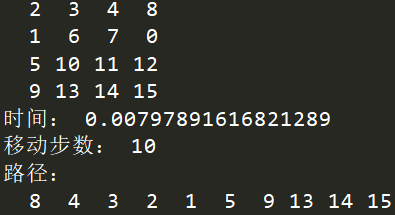
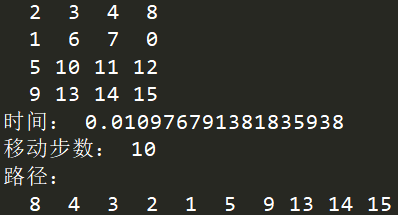
样例4 样例2

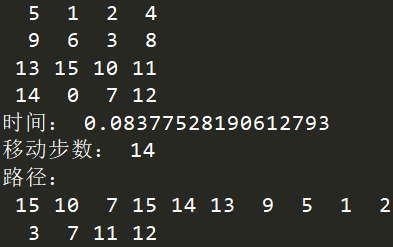
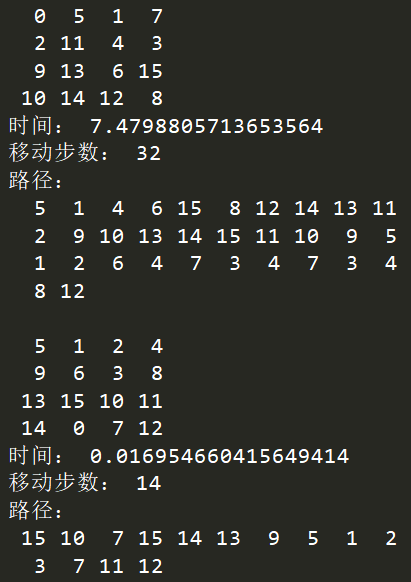


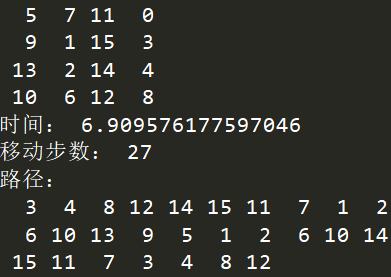
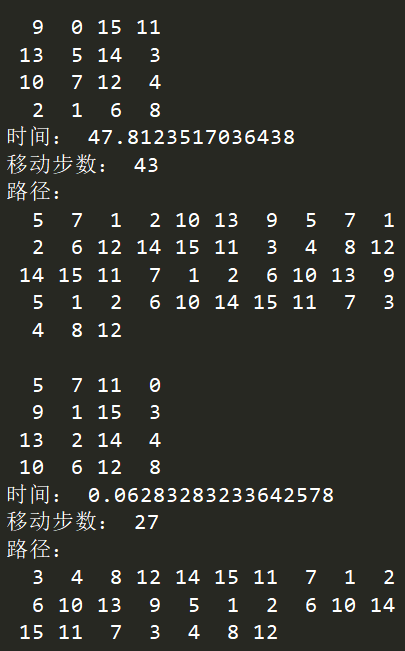


**欧氏距离（IDA\* 算法）：**

欧式距离 曼哈顿距离

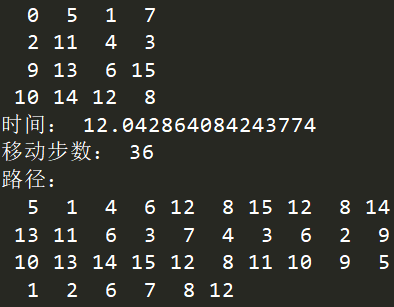
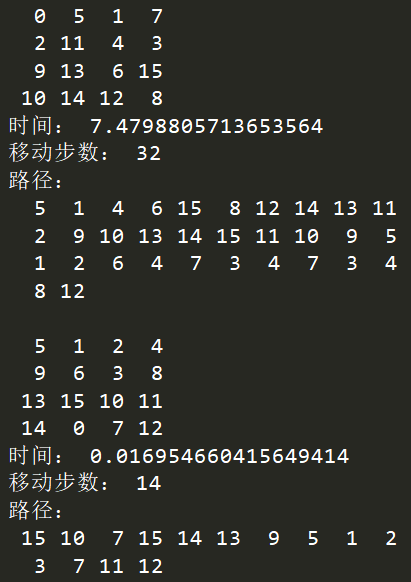
 

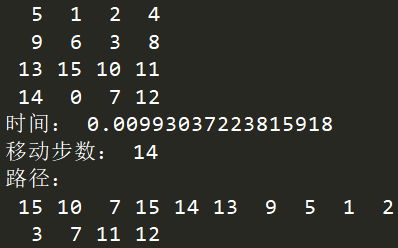
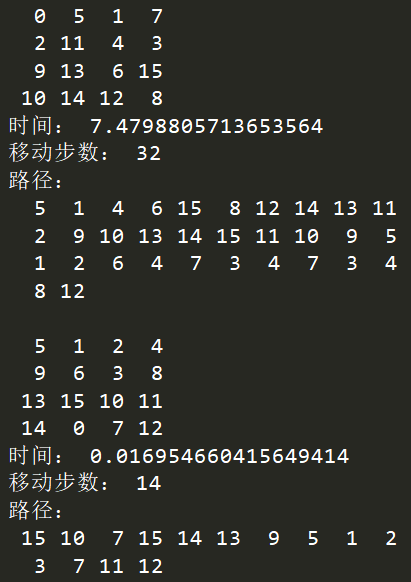
 

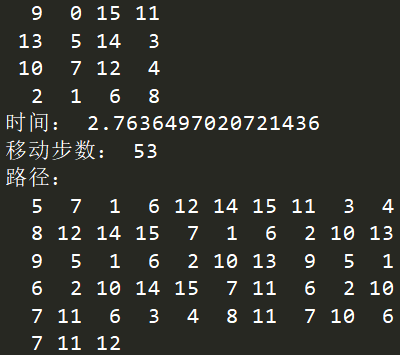
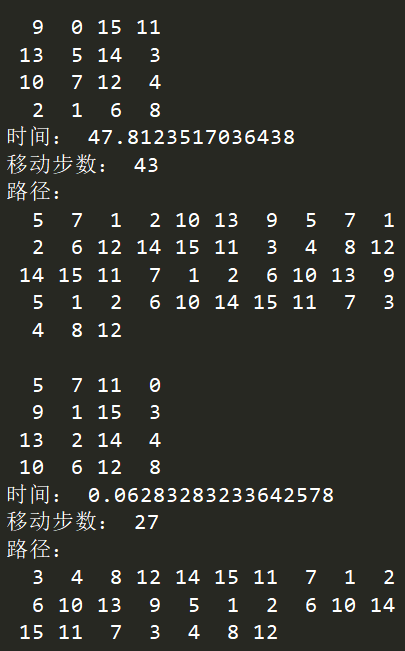
从上图中可以看出，使用欧氏距离作为启发式函数能找到最优路径，运行的时间多于使用曼哈顿距离所运行的时间。当总的移动步数大于30时，使用欧式距离的时间远大于使用曼哈顿距离使用的时间。

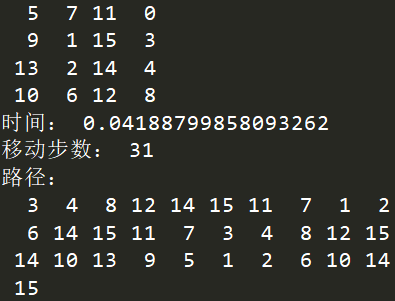
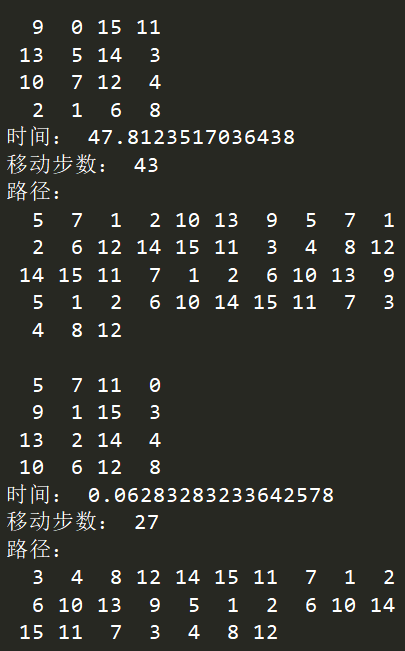
**1.5倍曼哈顿距离（IDA\* 算法）：**

1.5倍曼哈顿距离 曼哈顿距离

从上图可以看出1.5倍曼哈顿距离可以明显加快IDA\* 算法运行的速度，但它并不能确保可以找到最优路径。

* **评测指标展示**

使用曼哈顿距离作为启发式函数时，h(x)满足，预估值小于真实值，即可采纳性，h(x)也满足，即单调性，所以使用该启发式函数时可以找到一条从起始节点到目标节点的最短的路径。

使用欧式距离作为启发式函数时，h(x)满足，预估值小于真实值，即可采纳性，h(x)也满足，即单调性，所以使用该启发式函数时可以找到一条从起始节点到目标节点的最短的路径。

使用1.5倍曼哈顿距离作为启发式函数时，h(x)并不满足，即不满足采纳性，所以使用该启发式函数时不一定能找到一条从起始节点到目标节点的最短的路径。

设表示使用曼哈顿距离时的启发式函数，表示使用欧氏距离时的启发式函数，对于所有的节点满足，所以更具有优势，探索的节点比h2(x)更少，故使用曼哈顿距离所运行的时间小于使用欧氏距离所运行的时间。