

20337270\_钟海财\_实验3

中山大学计算机学院

本科生实验报告

(2021学年春季学期)

课程名称: Artificial Intelligence

教学班级            20级软工+网安  
学号                20337270

专业 (方向)  
姓名

软件工程  
钟海财

## lab3.1 盲目搜索

### 一、 实验题目

对于样例MazeData.txt:

**类型一:**

BFS、DFS都要实现

**类型二:**

迭代加深和双向搜索选一个实现，这里我选择实现双向搜索。

### 二、 实验内容

#### 1. 算法原理

每种搜索算法都有一个共同属性，即维护两个列表：开放列表（OPEN）和封闭列表（CLOSE）。

开放列表：包含树中所有等待搜索（或扩展）的节点。

封闭列表：包含所有已经探索过的节点和不再考虑的节点。

深度优先搜索DFS是用栈表示开放列表，栈是先进后出的数据结构。

广度优先搜索BFS是用队列表示开放列表，队列是先进先出的数据结构。

## BFS

BFS 是从根节点开始，沿着树(图)的宽度遍历树(图)的节点。如果所有节点均被访问或找到目标节点，则算法中止。一般用队列数据结构来辅助实现 BFS 算法。

## DFS

DFS 是从根节点开始，沿着树(图)的深度遍历树(图)的节点。如果所有节点均被访问或找到目标节点，则算法中止。一般用栈数据结构来辅助实现 DFS 算法。

## 双向搜索

从两个点同时分别依次进行BFS，如果所有节点均被访问或两个队列中有一个交点时算法中止，且两个队列中有一个交点时说明两个点是连通的，再从交点分别向两端回溯找到路径。

## 2. 伪代码/流程图

### 伪代码：

#### BFS

```
def bfs(起点S, 终点E, 将要访问的可达点near, 图data, 可达点ready, 父亲节点father):
    将起点S记录到可达点ready中
    for i in range(4): # 对S的4个邻接点进行判断
        p[i] = S的第i个邻接点
        if P[i]在图data中可达 and P[i]未被记录到ready中 then
            把P[i]记录到ready中表示已经可达
            把P[i]的父亲节点记录为S #father[p[i]] = S
            将P[i]记录到将要访问的可达点near中(near使用队列，做入队操作)
            if 终点已被记录到ready中 then
                return
    if 将要访问的可达点near为空:
        return
    将near的队首作为下一个要进行bfs的节点new_S # new_S = near.front()
    near的队首出队
    对下一个节点new_S进行bfs # bfs(new_S, 终点E, 将要访问的可达点near, 图data, 可达点
    ready, 父亲节点father):
        return
```

#### DFS

```
def dfs(起点S, 终点E, 路径stack, 图data, 可达点ready, 父亲节点father):
    将起点S记录到可达点ready中
    for i in range(4): # 对S的4个邻接点进行判断
        p[i] = S的第i个邻接点
        if P[i]在图data中可达 and P[i]未被记录到ready中 then
            把P[i]记录到ready中表示已经可达
            把P[i]的父亲节点记录为S #father[p[i]] = S
            将P[i]记录到路径stack中(stack使用栈，做入栈操作)
            if 终点已被记录到ready中 then
```

```

        return
    将路径stack的栈顶作为下一个要进行dfs的节点new_S    # new_S = stack.top()
    对下一个节点new_S进行dfs # dfs(new_S, 终点E, 路径stack, 图data, 可达点
ready, 父亲节点father):
    if 终点已被记录到ready中 then
        return
    路径stack的栈顶出栈
return

```

## 双向搜索

```

# 双向搜索,对队列1 near1的队首进行搜索
def bilateral(队列1 near1, 队列2 near2, 图data, 可达点ready, 队列1的可达点ready1, 队列2
的可达点ready2, is_find, father):
    将队列1 near1的队首作为将要进行搜索的节点S    # S = near1.front()
    near1的队首出队
    for i in range(4): # 对S的4个邻接点进行判断
        p[i] = S的第i个邻接点
        if P[i]在图data中可达 and P[i]未被记录到ready1中 then
            把P[i]记录到ready1和ready中表示已经可达
            把P[i]的父亲节点记录为S #father[p[i]] = S
            将P[i]记录到将要访问的可达点near1中(near1使用队列, 做入队操作)
            if P[i]已被记录到ready2中 then
                将S和P[i]记录到is_find中
                return
    if near1为空 or near2为空:
        return
    交换near1和near2的位置, 交换ready1和ready2的位置, 从另一个点的方向开始搜索
    # bilateral(near2, near1, 图data, 可达点ready, ready2, ready1, is_find, father)
    return

def bilateral_search(...):
    使用bilateral(...)双向搜索得到交点并存放到is_find中
    从交点1向一个方向回溯路径存放到the_way中
    从交点2向另一个方向回溯路径存放到the_way中
    输出路径the_way

```

## 3. 关键代码展示（带注释）

### DFS

```

def dfs(sx, sy, ex, ey, near_x, near_y, data, ready, father):    # 使用栈中的top, pop函数
    for i in range(4):
        x = sx + d_xy[0][i]
        y = sy + d_xy[1][i]
        if data[x][y] != '1' and ready[x][y] == 1:
            ready[x][y] = 0
            near_x.push(x)
            near_y.push(y)
            father[x][y] = [sx, sy]    # 用 father[x][y] = [sx,sy]来回溯路径
            if ready[ex][ey] == 0:    # dfs找到的路径就是在栈中
                return

```

```

        dfs(x, y, ex, ey, near_x, near_y, data, ready, father)
    if ready[ex][ey] == 0:
        return
    near_x.pop()
    near_y.pop()
return

```

## BFS

```

def bfs(sx, sy, ex, ey, near_x, near_y, data, ready, father):    # 使用队列中的
deque, front函数
    ready[sx][sy] = 0
    for i in range(4):
        x = sx + d_xy[0][i]
        y = sy + d_xy[1][i]
        if data[x][y] != '1' and ready[x][y] == 1:
            ready[x][y] = 0    # 用 father[x][y] = [sx,sy]来回溯路径
            father[x][y] = [sx, sy]
            if ready[ex][ey] == 0:
                return
            near_x.push(x)
            near_y.push(y)
    if near_x.empty():
        return
    new_x = near_x.front()
    new_y = near_y.front()
    near_x.dequeue()
    near_y.dequeue()
    bfs(new_x, new_y, ex, ey, near_x, near_y, data, ready, father)
return

```

## 双向搜索

```

def bilateral(near_x1, near_y1, near_x2, near_y2, data, ready, ready1, ready2,
is_find, father):    # 双向搜索
    sx = near_x1.front()    # 使用队列，双向都是bfs
    sy = near_y1.front()
    near_x1.dequeue()
    near_y1.dequeue()
    if data[sx][sy] == '1':    # 这句条件应该没用，处于两个队列中的(sx,sy)都是可达的
        return
    if ready1[sx][sy] == 0 and ready2[sx][sy] == 0:    # 这句一般没用，一般都是在产生
一个新的可达点进行判断
        return
    for i in range(4):
        x = sx + d_xy[0][i]
        y = sy + d_xy[1][i]
        if data[x][y] != '1' and ready1[x][y] == 1:    # 未到达过的可达点
            ready1[x][y] = 0
            ready[x][y] = 0
            if ready1[x][y] == 0 and ready2[x][y] == 0:    # 如果从两个点出发都能到达
该点，说明连通
                is_find[0] = True
                is_find[1] = x    # 这种情况不用令用 father[x][y] = [sx,sy]，分别从
两个点回溯路径
                is_find[2] = y

```

```

        is_find[3] = sx
        is_find[4] = sy
        return
    # 用 father[x][y] = [sx,sy]
    father[x][y] = [sx, sy]
    near_x1.push(x)
    near_y1.push(y)
    if near_x1.empty() or near_x2.empty():      # 如果满足该条件, 说明这两个点之间是不连
通的, 直接返回
        return
    bilateral(near_x2, near_y2, near_x1, near_y1, data, ready, ready2, ready1,
is_find, father)
    # 交换位置, 从另一个点开始搜索
    return

```

## 代码:

```

import time
# d_xy = [[0, 0, 1, -1], [-1, 1, 0, 0]] # 改变方向对bfs广度优先没什么影响, 但对dfs深度优
先影响很大
d_xy = [[1, 0, -1, 0], [0, -1, 0, 1]]

# 栈和队列共用一个类, 使用时调用不同函数, 栈使用pop, top; 队列使用dequeue, front, 其余
push, empty, size共用
class StackAndQueue:
    data = []

    def __init__(self):
        self.data = []

    def push(self, element):
        self.data.append(element)

    def dequeue(self): # 出队, 队首出队, 如果用切片操作是生成一个新列表self.data =
self.data[1:]
        self.data.pop(0)

    def pop(self):      # 出栈, 栈顶出栈, 如果用切片操作是生成一个新列表self.data =
self.data[:-1]
        self.data.pop(self.size()-1)

    def empty(self):
        return len(self.data) == 0

    def top(self): # 栈顶
        return self.data[-1]

    def front(self): # 队首
        return self.data[0]

    def size(self):
        return len(self.data)

def dfs(sx, sy, ex, ey, near_x, near_y, data, ready, father): # 使用栈中的top, pop函
数
    for i in range(4):
        x = sx + d_xy[0][i]

```

```

y = sy + d_xy[1][i]
if data[x][y] != '1' and ready[x][y] == 1:
    ready[x][y] = 0
    near_x.push(x)
    near_y.push(y)
    father[x][y] = [sx, sy]
    if ready[ex][ey] == 0:      # dfs找到的路径就是在栈中
        return
    dfs(x, y, ex, ey, near_x, near_y, data, ready, father)
    if ready[ex][ey] == 0:      # dfs找到的路径就是在栈中
        return
    near_x.pop()
    near_y.pop()
return

```

**def bfs**(sx, sy, ex, ey, near\_x, near\_y, data, ready, father): # 使用队列中的  
**dequeue**, **front**函数

```

ready[sx][sy] = 0
for i in range(4):
    x = sx + d_xy[0][i]
    y = sy + d_xy[1][i]
    if data[x][y] != '1' and ready[x][y] == 1:
        ready[x][y] = 0      # 用 father[x][y] = [sx,sy]
        father[x][y] = [sx, sy]
        if ready[ex][ey] == 0:
            return
        near_x.push(x)
        near_y.push(y)
if near_x.empty():
    return
new_x = near_x.front()
new_y = near_y.front()
near_x.dequeue()
near_y.dequeue()
bfs(new_x, new_y, ex, ey, near_x, near_y, data, ready, father)
return

```

**def bilateral**(near\_x1, near\_y1, near\_x2, near\_y2, data, ready, ready1, ready2,  
**is\_find**, father): # 双向搜索

```

sx = near_x1.front()      # 使用队列，双向都是bfs
sy = near_y1.front()
near_x1.dequeue()
near_y1.dequeue()
if data[sx][sy] == '1':    # 这句条件应该没用，处于两个队列中的(sx,sy)都是可达的
    return
if ready1[sx][sy] == 0 and ready2[sx][sy] == 0:    # 这句一般没用，一般都是在产生
一个新的可达点进行判断
    return

```

```

for i in range(4):
    x = sx + d_xy[0][i]
    y = sy + d_xy[1][i]
    if data[x][y] != '1' and ready1[x][y] == 1:    # 未到达过的可达点
        ready1[x][y] = 0
        ready[x][y] = 0
        if ready1[x][y] == 0 and ready2[x][y] == 0:    # 如果从两个点出发都能到达

```

该点，说明连通

```

is_find[0] = True

```

```

is_find[1] = x      # 这种情况不用令用 father[x][y] = [sx,sy]，分别从

```

两个点回溯路径

```

        is_find[2] = y
        is_find[3] = sx
        is_find[4] = sy
        return
    # 用 father[x][y] = [sx,sy]
    father[x][y] = [sx, sy]
    near_x1.push(x)
    near_y1.push(y)
    if near_x1.empty() or near_x2.empty():      # 如果满足该条件，说明这两个点之间是不连
通的，直接返回
        return
    bilateral(near_x2, near_y2, near_x1, near_y1, data, ready, ready2, ready1,
is_find, father)
    # 交换位置，从另一个点的方向开始搜索
    return

def out_answer(temp, sx, sy, ex, ey):
    for i in range(len(temp)):      # 输出路径上的所有点
        for j in range(len(temp[i])):
            if i == 0 or j == 0 or i == len(temp)-1 or j == len(temp[i])-1:      # 边界
                print("1", end="", sep="")
            elif i == sx and j == sy:
                print("S", end="", sep="")
            elif i == ex and j == ey:
                print("E", end="", sep="")
            elif temp[i][j] != 1:
                print(temp[i][j], end="", sep="")
            else:
                print(" ", end="", sep="")
        print("")
    return

def main():
    data = []      # 用于记录图
    sx, sy, ex, ey = -1, -1, -1, -1
    for line in open("MazeData.txt", "r"):      # 设置文件对象并读取每一行文件
        data.append(line[:-1])      # 将每一行文件加入到列表data中
    i = len(data)-1
    for j in range(len(data[i])):      # 找到起点和终点的位置
        if data[i][j] == "E":
            ex = i
            ey = j
        if data[i][j] == "S":
            sx = i
            sy = j
    time1 = time.time()
    len_x = len(data)
    len_y = len(data[0])
    # print(len_x, len_y)
    print("S=(", sx, ",", sy, ") E=(", ex, ",", ey, ")", sep="")      # 输出起点，终点的
位置

    ready = []      # 记录所有从起点出发可到达的点
    the_way = []      # 记录从起点出发到终点的路径上的点
    father = []      # 用于记录某点的父亲节点是谁，如father[x][y] = [fx][fy] ,表示是从
(fx,fy)出发访问到(x,y)
    for i in range(len_x):
        item = [1] * len_y
        ready.append(item)

```

```

        the_way.append(item[:])
        item2 = [[] * len_y
        father.append(item2)
ready[sx][sy] = 0    # 记录所有从起点出发可达的点，初始状态令起点S位置为0，表示S可达

near_x, near_y = StackAndQueue(), StackAndQueue() # 用于记录未访问的可达点的x,y坐标，依据bfs或dfs选择使用栈或队列
# 选择搜索方法，类型一：
dfs(sx, sy, ex, ey, near_x, near_y, data, ready, father)
# bfs(sx, sy, ex, ey, near_x, near_y, data, ready, father)
a, b = ex, ey
while len(father[ex][ey]) != 0:    # 从终点出发回溯路径上的点
    if a == sx and b == sy:
        break
    the_way[a][b] = 0
    fa = father[a][b][0]
    fb = father[a][b][1]
    a, b = fa, fb

# 以下为双向搜索用到的变量
near_x1, near_y1, near_x2, near_y2 = StackAndQueue(), StackAndQueue(),
StackAndQueue(), StackAndQueue()
ready1, ready2 = [], []
for i in range(len_x):
    item = [1] * len_y
    ready1.append(item)
    item2 = item[:]
    ready2.append(item2)
ready1[sx][sy] = 0
ready2[ex][ey] = 0 # 初始状态令起点S位置为0，表示S可达
near_x1.push(sx)
near_y1.push(sy)
near_x2.push(ex)
near_y2.push(ey)
is_find = [False, sx, sy, ex, ey]
# 类型二： 双向搜索
# bilateral(near_x1, near_y1, near_x2, near_y2, data, ready, ready1, ready2,
is_find, father)

for i in range(2):    # 从交点向两端回溯路径
    a, b = is_find[2*i+1], is_find[2*i+2]
    while True:
        if a == sx and b == sy or a == ex and b == ey:
            break
        the_way[a][b] = 0
        fa = father[a][b][0]
        fb = father[a][b][1]
        a, b = fa, fb
time2 = time.time()
if ready[ex][ey] == 0 or is_find[0]:    # 如果能从起点找到终点的话
    print("reachable")
    if is_find[0]: # 如果使用双向搜索找到了
        print(is_find[1], is_find[2])    # 输出双向搜索中相交点的位置
else:
    print("not reachable")
print("Used Time %f" % (time2 - time1), "sec")
temp = the_way[:]    # 关键路径上的点
# temp = ready[:]    # 所有到达过的节点
out_answer(temp, sx, sy, ex, ey)    # 输出路径

```



```
if __name__ == '__main__':  
    main()
```

#### 4. 创新点&优化 (如果有)

无

### 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）

每个节点的搜索方向方向为:  $d_{xy} = [[1, 0, -1, 0], [0, -1, 0, 1]]$ 时

DFS

[illegible]

## BFS

## 双向搜索

[illegible]

每个节点的搜索方向为:  $d_{xy} = [[0, 0, 1, -1], [-1, 1, 0, 0]]$ 时

DFS

[illegible]

**BFS**

[illegible]

## 双向搜索

[illegible]

## 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）

### 评测指标:

- 1.是否为最优路径
- 2.运行时间

### 分析：

DFS在两种节点的搜索方向下都不是最优路径，且不同的节点搜索方向对最终的路径都有较大的影响。且DFS的运行时间都是最短，说明在路径比较少的情況下DFS性能更好。

BFS和双向搜索无论在何种搜索方向下都是最优路径，且双向搜索的运行时间短于BFS，验证了

双向搜索的时间复杂度 $O(b^{\frac{d}{2}}) + O(b^{\frac{d}{2}}) < BFS$ 的时间复杂度 $O(b^d)$

## 结论：

## BFS 算法评估

(设分支因子为  $b$ ，最浅的目标结点的深度为  $d$ )：

完备性：当树的分支因子  $b$  是有限的，则算法是完备的。

最优性：如果路径代价是基于结点深度的非递减函数，则算法是最优的，否则不具备最优性。

复杂性：显然时空复杂度均为  $O(b^d)$ 。

适用于寻找最优路径。

## DFS 算法评估

(设分支因子为  $b$ ，最浅的目标结点的深度为  $d$ ，最深的叶子结点为  $m$ )：

完备性：对于树搜索，DFS 总是不完备的；而对于图搜索，DFS 在有限状态空间下是完备的，而在无限状态空间下是不完备的

最优性：显然不是最优的

复杂性：时间复杂度为  $O(bm)$ ，空间复杂度为  $O(bm)$ ，可见其空间复杂度比另外两种搜索小很多，这也是深度优先搜索最大的优势。

适用于较大的图的搜索。

## 双向搜索

完备性：当树的分支因子  $b$  是有限的，则算法是完备的。

最优性：如果路径代价是基于结点深度的非递减函数，则算法是最优的，否则不具备最优性。

复杂性：时空复杂性都是  $b^{d/2} + b^{d/2}$  要远小于  $b^d$ 。

适用于起点和终点都明确的时候。

## 四、思考题

本次实验无思考题。

## 五、参考资料

1. 实验文档：盲目搜索.pdf
2. 课本：《人工智能》(第三版) 清华大学出版社
3. 课件：03\_搜索技术.ppt

# lab3.2 启发式搜索

## 一、实验题目

利用A\*搜索算法和IDA\*搜索算法解决15-puzzle问题

实验测例包括(以列表表示):

测试样例1~6:

[1, 2, 4, 8, 5, 7, 11, 10, 13, 15, 0, 3, 14, 6, 9, 12]	# 22 (步数)
[5, 1, 3, 4, 2, 7, 8, 12, 9, 6, 11, 15, 0, 13, 10, 14]	# 15
[14, 10, 6, 0, 4, 9, 1, 8, 2, 3, 5, 11, 12, 13, 7, 15]	# 49
[6, 10, 3, 15, 14, 8, 7, 11, 5, 1, 0, 2, 13, 12, 9, 4]	# 48
[11, 3, 1, 7, 4, 6, 8, 2, 15, 9, 10, 13, 14, 12, 5, 0]	# 52+
[0, 5, 15, 14, 7, 9, 6, 13, 1, 2, 12, 10, 8, 11, 4, 3]	# 56+

新增测试样例1~3

[1, 15, 7, 10, 9, 14, 4, 11, 8, 5, 0, 6, 13, 3, 2, 12]	# 40
[1, 7, 8, 10, 6, 9, 15, 14, 13, 3, 0, 4, 11, 5, 12, 2]	# 40
[5, 6, 4, 12, 11, 14, 9, 1, 0, 3, 8, 15, 10, 7, 2, 13]	# 40

## 二、实验内容

### 1. 算法原理

本次实验主要针对两种启发式搜索进行实现 A\*搜索算法和 IDA\*搜索算法。

启发式搜索与盲目搜索或者无信息搜索最大的区别就在于启发式搜索采用了启发信息来引导整个搜索的过程，能够减少搜索的范围，降低求解问题的复杂度。这里的启发信息主要由估价函数组成，估价函数 $f(x)$ 由两部分组成，从初始结点到当前结点 $n$ 所付出的实际代价 $g(n)$ 和从当前结点 $n$ 到目标结点的最优路径的代价估计值 $h(n)$ ，即

$$f(n) = g(n) + h(n)$$

### 启发式函数设计:

启发式函数需要满足以下两种性质:

#### 1.可采纳的 (admissible)

定义 $h(n)$ 为到终点代价的预估值， $h^*(n)$ 为真实的代价  
若 $h(n) \leq h^*(n)$ 成立，即预估值小于真实值的时候，  
算法必然可以找到一条最短路径若 $h(n) \leq h^*(n)$ 成立，

## 2.单调的 (consistent)

$$h(n) \leq h(n') + cost(n, n')$$

### 例如：曼哈顿距离

在这个启发式函数中，利用曼哈顿距离公式

$$M = abs(node.x - goal.x) + abs(node.y - goal.y)$$

对除了数码块0以外的所有数码块计算其曼哈顿距离，并将其累加作为当前状态的总曼哈顿距离，

注：计算曼哈顿距离时之所以排除数码块0，是因为若考虑数码块0，则不满足可采纳性(Admissible)

## A\*算法和IDA\*算法的原理

### I. A\*算法

A\*搜索算法与BFS类似，但它需要加入估价函数作为启发信息。

#### 算法步骤

开始：

1.从起点A开始，将其当作待处理的结点加入open表中

2.将其子结点也加入open表中，计算f(x), g(x),h(x)，其中f(x)=g(x)+h(x)，g(x)表示当前结点的代价，h(x)为当前结点到达目标的代价的估计

3.从open表中取出结点A，并将其加入close表中，表示已经扩展过

循环：

4.从open表中弹出f(x)最小的结点C，并加入close表中

5.将C的子结点加入到open表中

6.如果待加入open表中的结点已经在open表中，则更新它们的g(x)，保持open表中该结点的g(x)最小

### II. IDA\*算法

#### 算法步骤：

算法基于深度优先搜索的思想，加入启发信息，对于给定的阈值bound，定义递归过程：

1. 从开始结点C，计算所有子结点的f(x)，选取最小的结点作为下一个访问结点
2. 对于某个节点，如果其f(x)大于bound，则返回当前结点的f(x)，取当中最小的f(x)对bound进行更新
3. 进行目标检测

## 2. 伪代码



## A\*算法

```
def A_star(open_list, close_list, other):    # A*算法
    find = False    # 记录是否找到目标节点
    t0 = 1    # 记录节点的总数 t0=len(open_list)+len(close_list)
    while True:
        p2 = open_list中f(x)最小的节点    # open_list[0]
        list0 = 生成p2的儿子节点
        for i in list0:
            p3 = 儿子节点list0[i]
            if 儿子节点p3不在open表 and 儿子节点p3不在close表 then    # 说明p3是新的节点
                ++t0    # 总结点数+1
                在节点p3中记录父亲节点p2在close表中的位置    # p3.father = len(close_list)
                将p3加入到open_list中(使用插入法同时完成排序)
                if p3是目标状态 then    # 目标状态即p3的h(x)==0
                    find = True
                    将节点p2从open表中去除
                    将节点p2, p3依次加入到close表
                    跳出循环
                end if
            end if
        end for
        将节点p2从open表中去除, 并加入到close表
    end while
    记录是否找到    # other.append(find)
    记录节点的总个数    # other.append(t0)
    return

def main():    # 主函数
    p1 = 测试样例
    open_list = []
    close_list = []
    other = []    # find, t0, 记录查找结果和节点总数
    将起始节点p1加入open表中 # open_list.append(p1)
    使用A*算法    # A_star(open_list, close_list, other)
    输出节点的总个数
    输出路径
```

## IDA\*算法

```
def IDA_search(path, bound):    # 对path进行最大深度为bound的dfs, 路径path使用栈
    p1 = 路径path的栈顶    # p1 = path[-1]
    if 如果p1的f(n)的值大于bound then
        return p1.f_x    # 则返回f(n)
    if 如果p1的h(n)的值 == 0:    # 根据h(n)进行目标检测
        return 0    # 返回0, 表示找到目标
    Min = 99999作为返回值, 是最小的大于本次bound的返回值, 作为下次深搜的bound
    list0 = p1的儿子节点
    for i in list0:
        将儿子节点 list0[i]压栈进路径path    # path.append(p3)
        再对路径path进行最大深度为bound的dfs, t = IDA_search(path, bound)
        if 如果得到的返回值t为0 then    # 如果得到的返回值为0, 表示找到目标, 迭代结束
            return 0
```

```

    end if
    if 如果返回值t<Min then
        对Min进行更新，取值最小的返回值作为Min # 使下次深搜的bound能最小
        Min = t
    end if
    路径栈去顶 # path.pop()
end for
return Min # 返回最小的Min作为下次深搜的bound，

```

```

def IDA_star(start): # 对起点start进行迭代加深搜索
    bound = 起点start的f(n)
    path = [start] # 路径集合，使用栈，开始的路径中只有起点
    while True:
        对path进行最大深度为bound的深搜，返回值ans # ans = IDA_search(path, bound)
        if 返回值ans为0 then # 说明找到了目标
            return 路径path
        end if
        用返回值ans对bound进行更新 # bound = ans
    end while
    return []

```

```

def main():
    p1 = 测试样例
    path = []
    对p1使用IDA*算法，路径保存到path中 # path = IDA_star(p1)
    输出路径

```

### 3. 关键代码展示（带注释）

#### A\*算法

```

import copy
import time
d_xy = [[0, 1, 0, -1], [1, 0, -1, 0]]
# d_xy = [[1, 0, -1, 0], [0, -1, 0, 1]]
goal = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]

class Node: # 节点的类
    father = -1 # 记录父节点在close表中的位置，起始节点的父亲节点设为-1
    data = [] # 状态列表data，将二维列表转化为一维列表储存
    last_key = 0 # 记录父节点的状态，避免生成的子节点中包含父节点
    key_0 = 0 # 记录当前状态中0在一维列表中的位置，
    # 转化为二维列表的规则：x0 = key_0%4，y0 = key_0//4；key = x +
    y*4
    f_x = 0 # 估价函数，f(x)=h(x)+g(x)
    g_x = 0 # 深度
    h_x = 0 # 启发函数，采用曼哈顿距离

    def __init__(self, list1, last, g_x1): # 通过给出“一个状态列表，父亲节点的位置，深度”构造一个节点
        self.g_x = g_x1 # 该节点的深度g(x)
        self.last_key = last # 记录父节点中0的位置
        self.data = list1[:]
        num = 0

```

```

for i in range(16):
    if self.data[i] == 0:
        self.key_0 = i
    else:
        x1 = i % 4
        y1 = i // 4
        x2 = (self.data[i] - 1) % 4
        y2 = (self.data[i] - 1) // 4
        d = abs(x2 - x1) + abs(y2 - y1)
        num += d
self.h_x = num          # h(x), 曼哈顿距离
self.f_x = num + self.g_x  # h(x)+g(x)

```

`def get_son(key0, last_key):` # 返回儿子节点的索引,将二维列表中的位置转化为一维列表中的位置

```

x0 = key0 % 4
y0 = key0 // 4
next_key = []
for i in range(4):
    x = x0 + d_xy[0][i]
    y = y0 + d_xy[1][i]
    if 0 <= x <= 3 and 0 <= y <= 3:
        key = x + y * 4
        if key != last_key: # 避免由子节点返回父节点
            next_key.append(key)
return next_key

```

`def is_in(node, list1):`

```

for i in range(len(list1)): # 可用二分查找法或其他优化
    if node.data == list1[i].data:
        return i
return -1

```

`def out_answer(close_list):` # 输出关键路径

```

f = len(close_list)-1
the_way = []
# 回溯找关键路径,从close表最后一条开始,逐个查找其父亲节点,
# 直到某节点父亲节点为-1(即起始节点),并将所有查到的序列写入the_way并输出
while True:
    the_way.append(close_list[f])
    f = close_list[f].father
    if f == -1:
        break
t = len(the_way)
for k in range(t): # 输出关键路径
    print("move:", k, " h(x):", the_way[t-k-1].h_x, " g(x):", the_way[t-k-1].g_x, " f(x):", the_way[t-k-1].f_x)
    for i in range(16):
        q = the_way[t-k-1].data[i]
        print(q, end=" ")
        if q < 10:
            print(" ", end="")
        if (i+1) % 4 == 0:
            print("")
    print("")

```

`def A_star(open_list, close_list, other):`

```

find = False
t0 = 1 # 记录节点的总数=len(open_list)+len(close_list)
while True:
    p2 = copy.deepcopy(open_list[0])
    key0 = p2.key_0
    list0 = get_son(p2.key_0, p2.last_key)
    for i0 in range(len(list0)):
        new_list = p2.data[:] # 进行移动，即交换0与周围的位置
        tp = new_list[list0[i0]]
        new_list[list0[i0]] = new_list[key0]
        new_list[key0] = tp
        p3 = Node(new_list, p2.key_0, p2.g_x+1) # 生成p3
        a = is_in(p3, open_list)
        b = is_in(p3, close_list)
        if a != -1: # 进行更新，似乎没必要，因为后续节点的g(x)更大，而h(x)相
            if p3.f_x < open_list[a].f_x:
                open_list[a].f_x = p3.f_x
                open_list[a].g_x = p3.g_x
                open_list[a].h_x = p3.h_x
            if a == -1 and b == -1: # 说明是新的节点
                p3.father = len(close_list) # 在节点p3中记录父亲节点p2在close表中的
            t0 += 1
            # open_list.append(p3)
            if len(open_list) == 0 or p3.f_x >= open_list[-1].f_x:
                open_list.append(p3)
            else:
                for i in range(1, len(open_list)):
                    if p3.f_x < open_list[i].f_x:
                        open_list.insert(i, p3) # 进行插入，完成排序
                        break
                if p3.h_x == 0: # 目标状态
                    find = True
                    open_list.pop(0)
                    close_list.append(p2)
                    close_list.append(p3)
                    break
            open_list.pop(0)
            if find or len(open_list) == 0:
                break
            close_list.append(p2)
    other.append(find) # 记录是否找到
    other.append(t0) # 记录节点的总个数
    return

```

同

位置

```

def main():
    print("start A*:")
    time1 = time.time()
    # 选择测试样例
    # p1 = Node([1, 2, 4, 8, 5, 7, 11, 10, 13, 15, 0, 3, 14, 6, 9, 12], -1, 0) #
    # p1 = Node([5, 1, 3, 4, 2, 7, 8, 12, 9, 6, 11, 15, 0, 13, 10, 14], -1, 0) #
    # p1 = Node([14, 10, 6, 0, 4, 9, 1, 8, 2, 3, 5, 11, 12, 13, 7, 15], -1, 0) # 49
    # p1 = Node([6, 10, 3, 15, 14, 8, 7, 11, 5, 1, 0, 2, 13, 12, 9, 4], -1, 0) #
    # p1 = Node([11, 3, 1, 7, 4, 6, 8, 2, 15, 9, 10, 13, 14, 12, 5, 0], -1, 0) #

```

52+

```

# p1 = Node([0, 5, 15, 14, 7, 9, 6, 13, 1, 2, 12, 10, 8, 11, 4, 3], -1, 0)      #
56+

# p1 = Node([1, 15, 7, 10, 9, 14, 4, 11, 8, 5, 0, 6, 13, 3, 2, 12], -1, 0)      #
40

# p1 = Node([1, 7, 8, 10, 6, 9, 15, 14, 13, 3, 0, 4, 11, 5, 12, 2], -1, 0)      #
40

p1 = Node([5, 6, 4, 12, 11, 14, 9, 1, 0, 3, 8, 15, 10, 7, 2, 13], -1, 0)      # 40
open_list = []
close_list = []
other = [] # find, t0
open_list.clear()
close_list.clear()
other.clear()
open_list.append(p1)
A_star(open_list, close_list, other)

time2 = time.time()
if other[0]:
    print("have answer!")
    out_answer(close_list)
    print("all puzzles", len(open_list)+len(close_list), other[1]) # 输出节点的
    总个数
    print("Used Time %f" % (time2 - time1), "sec")
else:
    print("no answer!")

if __name__ == '__main__':
    main()

```

## IDA\*算法

```

import copy
import time
d_xy = [[0, 1, 0, -1], [1, 0, -1, 0]]
# d_xy = [[1, 0, -1, 0], [0, -1, 0, 1]]
goal = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]

class Node: # 节点的类
    data = [] # 状态列表data, 将二维列表转化为一维列表储存
    last_key = 0 # 记录父节点的状态, 避免生成的子节点中包含父节点
    key_0 = 0 # 记录当前状态中0在一维列表中的位置,
    # 转化为二维列表的规则: x0 = key_0%4 , y0 = key_0//4 ; key = x +
    y*4
    f_x = 0 # 估价函数, f(x)=h(x)+g(x)
    g_x = 0 # 深度
    h_x = 0 # 启发函数, 采用曼哈顿距离

    def __init__(self, list1, last, g_x1): # 通过给出 “一个状态列表, 父亲节点的位置, 深度”构造一个节点
        self.g_x = g_x1 # 该节点的深度g(x)
        self.last_key = last # 记录父节点中0的位置
        self.data = list1[:]
        num = 0
        for i in range(16):
            if self.data[i] == 0:

```

```

        self.key_0 = i
    else:
        x1 = i % 4
        y1 = i // 4
        x2 = (self.data[i] - 1) % 4
        y2 = (self.data[i] - 1) // 4
        d = abs(x2 - x1) + abs(y2 - y1)
        num += d
    self.h_x = num          # h(x), 曼哈顿距离
    self.f_x = num + self.g_x  # h(x)+g(x)

```

`def get_son(key0, last_key):` # 返回儿子节点中0的位置,将二维列表中的位置转化为一维列表中的位置

```

x0 = key0 % 4
y0 = key0 // 4
next_key = []
for i in range(4):
    x = x0 + d_xy[0][i]
    y = y0 + d_xy[1][i]
    if 0 <= x <= 3 and 0 <= y <= 3:
        key = x + y * 4
        if key != last_key: # 避免由子节点返回父节点
            next_key.append(key)
return next_key

```

`def out_answer(path):` # 输出关键路径

```

t = len(path)
for k in range(t): # 输出关键路径
    print("move:", k, " h(x):", path[k].h_x, " g(x):", path[k].g_x, " f(x):",
path[k].f_x)
    for i in range(16):
        q = path[k].data[i]
        print(q, end=" ")
        if q < 10:
            print(" ", end="")
        if (i+1) % 4 == 0:
            print("")
    print("")

```

`def IDA_search(path, bound):`

```

p1 = path[-1]
if p1.f_x > bound: # 如果f(n)的值大于bound则返回f(n)
    return p1.f_x
if p1.h_x == 0: # 目标检测, 以0表示找到目标
    return 0
Min = 99999
p2 = copy.deepcopy(p1)
key0 = p2.key_0
list0 = get_son(p2.key_0, p2.last_key)
for i0 in range(len(list0)): # 生成儿子节点
    new_list = p2.data[:] # 进行移动, 即交换0与周围的位置
    tp = new_list[list0[i0]]
    new_list[list0[i0]] = new_list[key0]
    new_list[key0] = tp
    p3 = Node(new_list, key0, p2.g_x + 1) # 生成p3
    path.append(p3) # dfs压栈
    t = IDA_search(path, bound)
    if t == 0: # 如果得到的返回值为0, 表示找到目标, 迭代结束

```

```

        return 0
    if t < Min: # 如果返回值不是0, 且f>bound, 这时对Min进行更新, 取值最小的返回值作
为Min
        Min = t
        path.pop() # 栈去顶
    return Min

def IDA_star(start):
    bound = start.f_x # start.f_x # IDA*迭代限制,55
    path = [start] # 路径集合, 视为栈
    while True:
        ans = IDA_search(path, bound) # path, g, Hx, bound
        print(ans) # 输出深度, 查看深度变化
        if ans == 0:
            return path
        if ans == -1:
            return None
        bound = ans # 此处对bound进行更新
    return []

def main():
    print("start A*:")
    time1 = time.time()
    # 选择测试样例
    # p1 = Node([1, 2, 4, 8, 5, 7, 11, 10, 13, 15, 0, 3, 14, 6, 9, 12], -1, 0) #
22
    # p1 = Node([5, 1, 3, 4, 2, 7, 8, 12, 9, 6, 11, 15, 0, 13, 10, 14], -1, 0) #
15
    # p1 = Node([14, 10, 6, 0, 4, 9, 1, 8, 2, 3, 5, 11, 12, 13, 7, 15], -1, 0) #
49
    # p1 = Node([6, 10, 3, 15, 14, 8, 7, 11, 5, 1, 0, 2, 13, 12, 9, 4], -1, 0) #
48
    # p1 = Node([11, 3, 1, 7, 4, 6, 8, 2, 15, 9, 10, 13, 14, 12, 5, 0], -1, 0) #
52+
    # p1 = Node([0, 5, 15, 14, 7, 9, 6, 13, 1, 2, 12, 10, 8, 11, 4, 3], -1, 0) #
56+

    p1 = Node([1, 15, 7, 10, 9, 14, 4, 11, 8, 5, 0, 6, 13, 3, 2, 12], -1, 0) # 40
    # p1 = Node([1, 7, 8, 10, 6, 9, 15, 14, 13, 3, 0, 4, 11, 5, 12, 2], -1, 0) #
40
    # p1 = Node([5, 6, 4, 12, 11, 14, 9, 1, 0, 3, 8, 15, 10, 7, 2, 13], -1, 0) #
40

    path = []
    path.clear()
    path = IDA_star(p1)

    time2 = time.time()
    if len(path) > 0:
        print("have answer!\n")
        out_answer(path)
        print("all puzzles", len(path)) # 输出使用节点的最多个数
        print("Used Time %f" % (time2 - time1), "sec")
    else:
        print("no answer!\n")

if __name__ == '__main__':
    main()

```

## 4. 创新点&优化（如果有）

无

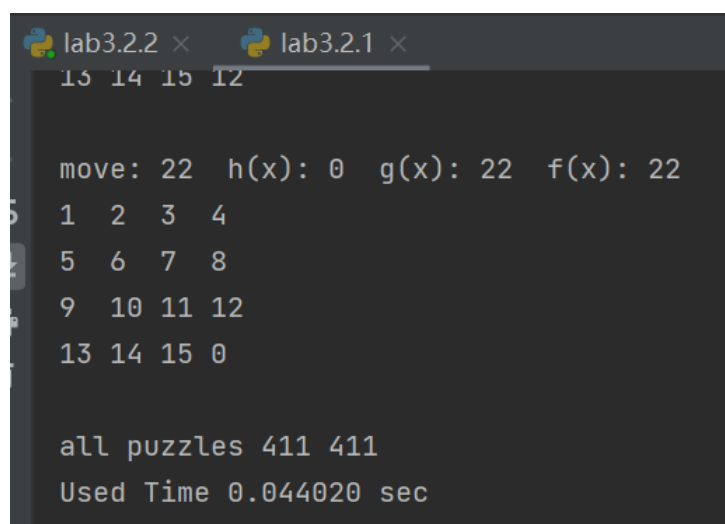
## 三、实验结果及分析

### 1. 实验结果展示示例（可图可表可文字，尽量可视化）

#### A\*算法

只能通过基本测试样例中的1、2，剩下的基本测试样例中的3、4和挑战性测试样例中的1、2都过不了，能通过3个增加测试样例。

#### 基本测试样例1



```
lab3.2.2 x lab3.2.1 x
13 14 15 12

move: 22 h(x): 0 g(x): 22 f(x): 22
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0

all puzzles 411 411
Used Time 0.044020 sec
```

#### 基本测试样例2



```
lab3.2.1 ×  
  
move: 15  h(x): 0  g(x): 15  f(x): 15  
1  2  3  4  
5  6  7  8  
9  10 11 12  
13 14 15 0  
  
all puzzles 40 40  
Used Time 0.001030 sec
```

## 增加测试样例1

```
move: 40  h(x): 0  g(x): 40  f(x): 40  
1  2  3  4  
5  6  7  8  
9  10 11 12  
13 14 15 0  
  
all puzzles 17978 17978  
Used Time 40.188513 sec
```

## 增加测试样例2

```
move: 40  h(x): 0  g(x): 40  f(x): 40  
1  2  3  4  
5  6  7  8  
9  10 11 12  
13 14 15 0  
  
all puzzles 12686 12686  
Used Time 21.298539 sec
```

## 增加测试样例3

```
move: 40  h(x): 0  g(x): 40  f(x): 40
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0

all puzzles 6908 6908
Used Time 5.471394 sec
```

## IDA\*算法

只能通过基本测试样例中的1、2、3、4，剩下的挑战性测试样例中的1、2都过不了，能通过3个增加测试样例。

### 基本测试样例1

```
move: 22  h(x): 0  g(x): 22  f(x): 22
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0

all puzzles 23
Used Time 0.008733 sec
```

### 基本测试样例2

```
lab3.2.2 × lab3.2.test ×
move: 15  h(x): 0  g(x): 15  f(x): 15
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0

all puzzles 16
Used Time 0.004502 sec
```

### 基本测试样例3

```
move: 49  h(x): 0  g(x): 49  f(x): 49
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0

all puzzles 50
Used Time 457.905494 sec
```

### 基本测试样例4

```
move: 48  h(x): 0  g(x): 48  f(x): 48
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0

all puzzles 49
Used Time 313.129111 sec
```

### 增加测试样例1

```
move: 40  h(x): 0  g(x): 40  f(x): 40
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0

all puzzles 41
Used Time 0.855314 sec
```

### 增加测试样例2

```
move: 40  h(x): 0  g(x): 40  f(x): 40
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0

all puzzles 41
Used Time 0.252418 sec
```

增加测试样例3

```
lab3.2.2 x

move: 40  h(x): 0  g(x): 40  f(x): 40
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0

all puzzles 41
Used Time 0.307505 sec
```

结果统计表

运行时间							
运行时间/s	基本测试样例1	基本测试样例2	基本测试样例3	基本测试样例4	增加测试样例1	增加测试样例2	增加测试样例3
步数	22	15	49	48	40	40	40
A*算法	0.044020	0.001030	过长	过长	40.188513	21.298539	5.471394
IDA*算法	0.008733	0.044502	457.905494	313.129111	0.855314	0.252418	0.307505

运行过程中节点的最多数目							
节点最多数目/个	基本测试样例1	基本测试样例2	基本测试样例3	基本测试样例4	增加测试样例1	增加测试样例2	增加测试样例3
步数	22	15	49	48	40	40	40
A*算法	411	40	过长	过长	17978	12686	6908
IDA*算法(空间复杂度=O(bm))	22*3	15*3	49*3	48*3	40*3	40*3	40*3

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）

## 评测指标:

- 1.运行时间
- 2.运行过程中节点的最多数目(空间复杂度)

## 分析:

### 对于指标1.运行时间:

通过结果统计表可知,当步数较少如基本测试样例2(15步),即目标节点的深度较小时,使用A\*算法的运行时间较小于IDA\*算法。但此时无论是A算法还是IDA算法都能较快找到目标。

但当步数较多如增加测试样例1, 2, 3(40步),即目标节点的深度较大时,使用A\*算法的运行时间远大于IDA\*算法,而此时IDA\*算法明显优于A\*算法。而对于基本测试样例3、4,使用IDA\*算法需要经过较长时间才能找到目标,而使用A\*算法的时间过长都找不到目标。

### 对于指标2.运行过程中节点的最多数目(空间复杂度):

通过结果统计表可知,当目标节点的深度较小时,A\*算法的空间复杂度较小,而目标节点的深度较大时,A\*算法的空间复杂度明显远远增加;而IDA\*算法的空间复杂度时钟保持在较小的水平 $O(bm)$ .在空间复杂度方面,IDA\*算法明显优于A\*算法。

综上,目标节点的深度较小时,A\*算法的时间复杂度优于IDA\*算法,而目标节点的深度较大时,IDA\*算法的时间复杂度优于A\*算法。此外,IDA\*算法的空间复杂度始终优于A\*算法。

## 四、 思考题

本次实验无思考题。

## 五、 参考资料

1. 实验文档: 06\_启发式搜索.pdf
2. 课本:《人工智能》(第三版) 清华大学出版社
3. 课件: 03\_搜索技术.ppt