人工智能实验

A*与 IDA*解决 15-puzzle 问题

16 级计科教务 2 班

16337327

郑映雪

实验题目

A*与 IDA*解决 15-puzzle 问题

实验内容

算法原理

1、A*算法

A*算法是启发式搜索的算法，本身是宽度优先搜索的变形。关键步骤在于从当前搜索结点往下选择下一步结点的时候，可以通过一个启发式函数选择代价最少的结点作为下一步的搜索结点。宽度优先搜索是盲目型的搜索，A*算法通过设计一个启发函数，可以减少宽度优先搜索的搜索时间以及得到一个较优解。

A*算法最核心的部分在于估值函数的设计：

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

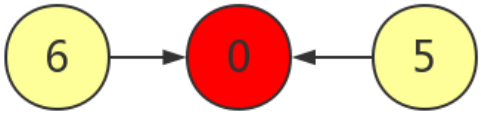
其中，g（n）为其实搜索点到当前点的代价（在这次实验中是搜索树的深度）；h（n）可以有多种设计方式，但需要满足单调性、可采纳性的条件，且 h（n）≤h*（n）（h（n）尽量接近 h*（n）为佳），才是一个合格的 h（n）函数，否则可能会找不到解。

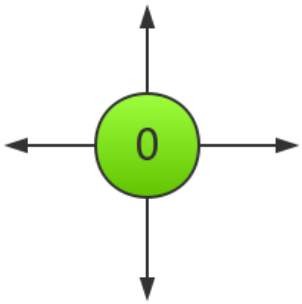
2、A*算法在本实验中的流程：

在本实验中，每一个结点代表当前矩阵的状态。设置两个列表模拟优先队列：open 表和 close 表。从起点开始，首先将起点存入 open 表，从起点搜寻上下左右四种状态作为子节点（前提是不出界）。为这些结点计算 f（n）。从 open 表中删除起点并把它加入到 close 表中,因为它已经不用再次搜索了。在 open 表剩下的结点里，选择 f（n）值最小的结点，重复扩展搜索的操作，当 open 表里已经有搜索到的结点时，更新 g（n）

的值。当 open 表为空时，代表不存在解；当 closed 表里已经有目标结点时，根据结点的父节点逐步往上搜索，可以输出路径。还有些特别的详细说明：

①探索的操作和结点的表示





左图是每一个操作步骤的示例——把空格的数字移到空格上。但这样的操作是繁琐的。所以我们可以把“移数字”变为右图所示的“移空格”。这样只需要对 0 的 x 值和 y 值进行操作即可。（回到上周的迷宫的操作）

结点的表示除了基本的特征外，最重要的是表示当前状态下的矩阵，这样可以判断矩阵是否与目标矩阵相等，从而判断当前状态是否为目标状态。

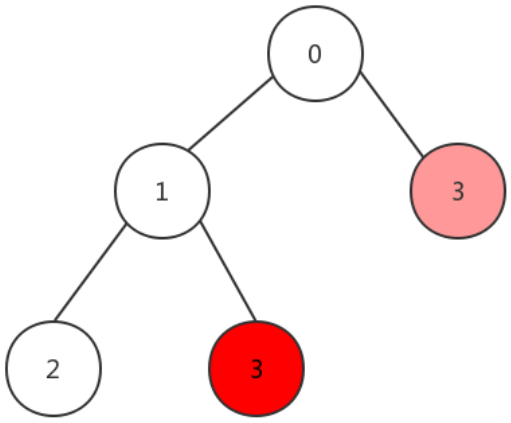
②f（n）的计算

f（n）=g（n）+h（n）。在这个实验中，g（n）可以表示为从起点到第 n 步移动的次数。h（n）可以选择矩阵中所有元素对目标位置的曼哈顿距离之和。（也可以表示成不在目标位置的数字个数）

3、IDA*算法

IDA*算法是迭代加深算法与 A*算法的结合。A*算法继承了宽度优先搜索的缺点——需要维护两个表而导致内存占用过大。实验证明，如果是简单一些的矩阵操作还是可以快速出来结果的，但是矩阵如果复杂了（比如 PPT 上的数据范例……），就卡死出不来。所以就有了 IDA*算法。IDA*算法结合了迭代加深搜索，采用深度优先搜索的模式，不需要维护很多表，只需要维护一个路径栈，所以大大节省了空间。但是由于迭代加深的特性，每一次在限制内搜索失败都要重头搜索。

如图所示：



迭代加深搜索在深度优先搜索的基础上进行了变形。它一次次限制可以搜索到的最多层数，逐层叠加，这样就可以保证自己找到目标节点的第一个路径是最优路径。即右子树的 3 为最优路径。

IDA*正是采用了这个办法，不仅省去了 A*的占用内存大的问题，而且可以保证第一个搜到的目标状态的路径是最优路径。

4、本实验中的 IDA*操作

IDA*的结点操作与 A*相同，这里就不赘述了。不同的是 A*维护了两个表，而 IDA*不需要存那么多结点，只需要存储搜索的路径的栈即可（由于回溯，栈的长度不会超过最优路径的长度，所以栈的规模是很小的）。但是，IDA*不是简简单单地套用之前写过的迭代加深搜索代码就行了，它也有变化：

它的变化是 k 值变 f（n）值。

迭代加深搜索限制的是层数，即每次最多搜索 k 层，如果搜索不到目标结点，则增加 k 的值。在 IDA*中，限制的是最大的 f（n），即，选取起点的 f（n）为初始的 maxf（n），开始搜索 f（n）值不大于 maxf(n)值的结点。当遇到超过 maxf（n）的结点时，则回溯，但要记录下这个结点的 f（n）值。当所有的结点达到边界后也没有解时，

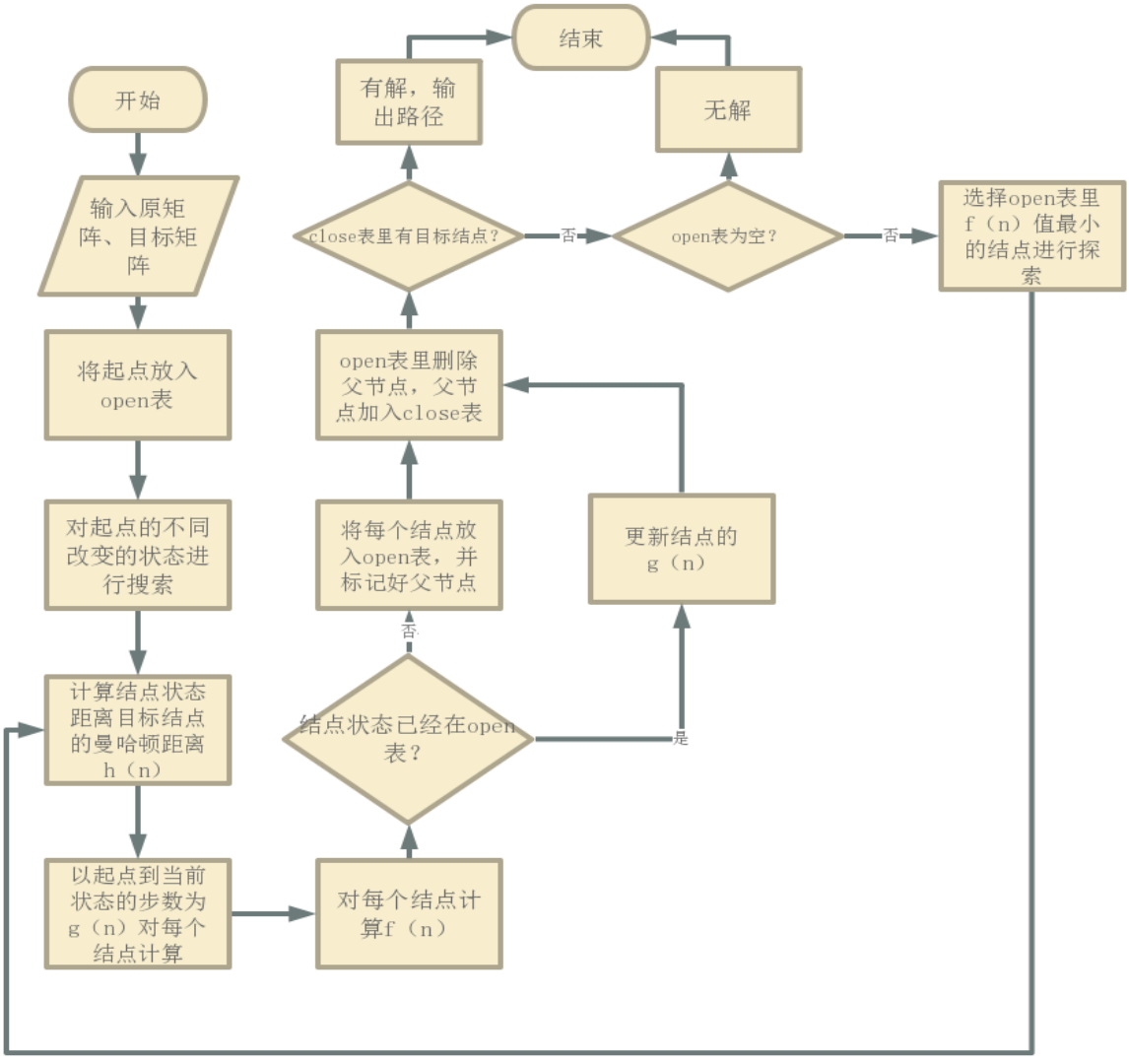
从起点开始重新搜索，同时 maxf（n）变为搜索中遇到的所有回溯结点（即 f（n）越界结点）中最大的 f（n）值，然后从起点开始下一轮搜索。

5、不同的 h(n)尝试

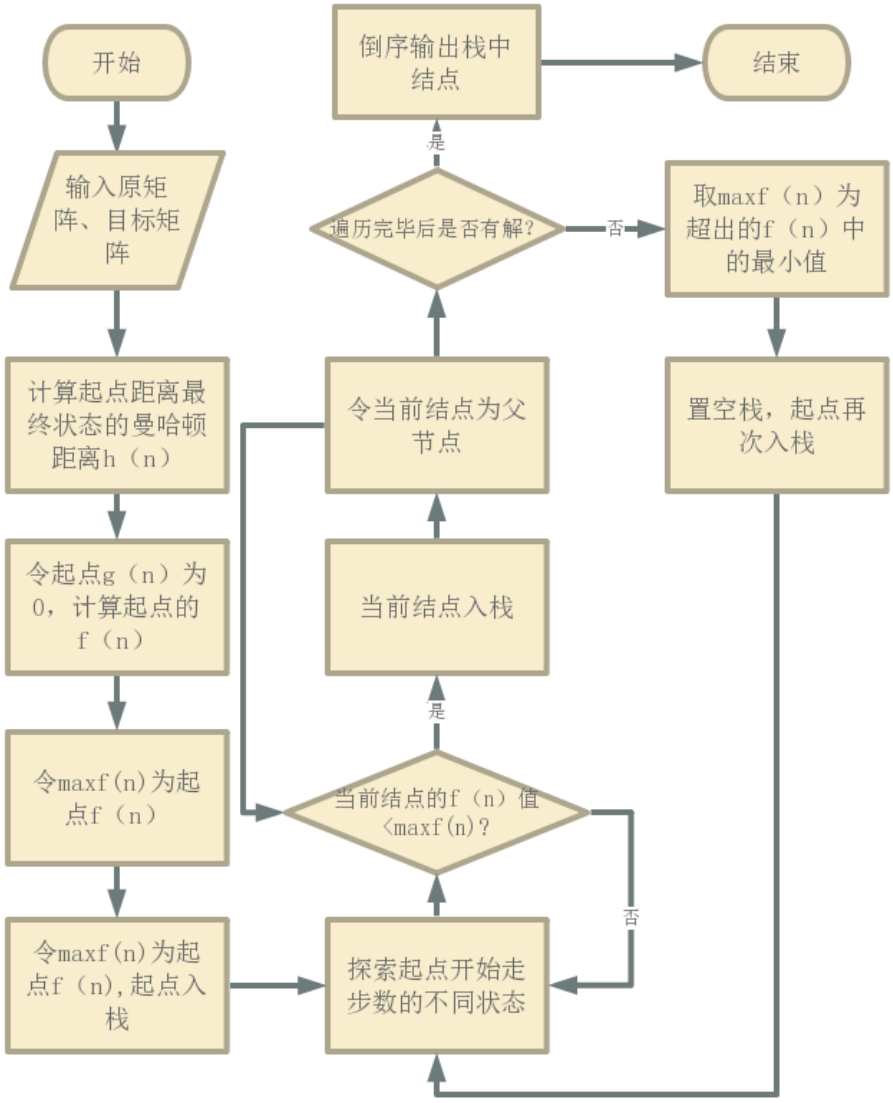
h（n）函数的选择对于一个算法来说也是很重要的，具体的前面我已经进行了阐述。而在本实验中我尝试用 2 种不同的 h（n）：一种是曼哈顿距离，一种是不在目标点上的数字个数。这两个 h（n）都满足条件。

流程图（以 h（n）为曼哈顿距离为例）

1、A*算法解决 15-puzzle 问题



2、IDA*算法解决 15-puzzle 问题



关键代码

1、数据准备

①结点设置

将每个结点设置为一个对象封装起来，囊括了搜索需要的的所有元素。

```
class node:
    def __init__(self, value=-1, x=-1, y=-1, currentdata=[], father=None,
gx=0, hx=0, fx=0):
```

```
        self.value = value#与 0 交换位置的数（可以作为路径输出）
        self.x = x#0 的坐标
        self.y = y
        self.currentdata = currentdata#当前状态矩阵
        self.father = father#父节点
        self.gx = gx
        self.hx = hx
        self.fx = fx
```

②做好计算曼哈顿距离的相关准备，包括目标距离的列表和根据输入矩阵找到 0 值所在坐标的搜索，以便建立起点。

```
goaldis = [(-1, -1)]
for i in range(4):
    for j in range(4):
        goaldis.append((i, j))#目标距离列表，包含所有目标结点的坐标，以便计算曼哈顿距离
for i in range(4):#找到 0 值所在的坐标
    for j in range(4):
        if data[i][j]==0:
            firstx=i
            firsty=j
x=node(-1,firstx,firsty,data)#起点建立
```

2、A*算法核心

①准备

设置 open、close 表，设置移动坐标矩阵和有无解判定。

```
behave = [[0, 1], [0, -1], [-1, 0], [1, 0]]  # 上下左右操作
open = []
close = []
flag = 0#是否找到解
```

起点建立，首先将起点加入 open 表。每次搜索前，从 open 表里找到 f（n）值最小的结点，以此结点再次展开搜索，直到有解或 open 表为空（无解）。

```
open.append(x)
tmpfather = None
k = 0
path = []
while flag == 0 and len(open) != 0:#还没找到解时
    minfx = 10000
    minindex = 0
```

```
    for i in range(len(open)):
        if open[i].fx < minfx:
            minfx = open[i].fx
            minindex = i#找到 f（n）最小的结点
    astar(open[minindex], open[minindex].currentdata, goaldis)#对它进行扩展
```

②搜索

对当前状态的下一步状态进行遍历，并分别计算 f（n），加入 open 表，同时从 open 表里删除父节点，在 close 表里加入父节点。

```
def astar(node1, data, goaldis):
    for i in range(4):
        newx = node1.x + behave[i][0]
        newy = node1.y + behave[i][1]
        newgx = node1.gx + 1
        if newx > 3 or newx < 0 or newy > 3 or newy < 0:#越界跳过
            continue
        newdata = copy.deepcopy(data)
        tmp = data[newx][newy]
        newdata[node1.x][node1.y] = tmp
        newdata[newx][newy] = 0
        newnode = node(tmp, newx, newy, newdata, node1, newgx)#建立新节点
        newhx = 0
        for i in range(4):
            for j in range(4):
                if newdata[i][j] != 0:
                    a, b = goaldis[newdata[i][j]]
                    newhx += abs(i - a) + abs(j - b)#计算曼哈顿距离
        newnode.hx = newhx
        newnode.fx = newnode.hx + newnode.gx#计算 f（n）
        for opens in open:
            if newnode.currentdata == opens.currentdata and opens.fx >
newnode.fx:
                opens.fx = newnode.fx#有重复值则更新值
        open.append(newnode)#加入 open 表
    open.remove(node1)#open 表里删除父节点
    close.append(node1)#父节点加入 close 表
```

③输出

找到解后根据父节点往上回溯直至找到起点，即为路径（省去了存储路径的大内存）。

```
while tmpfather != None and tmpfather.value != -1:#通过终点的父节点在 close 表里搜索，直至追溯到起点
    path.append(tmpfather.value)
    tmpfather = tmpfather.father
for i in range(len(path)):#倒序输出即为路径
    print(path[-1 - i],end=' ')
```

3、IDA*算法核心

①取起点为初始 maxfx，并在每次遍历完毕后以超过 maxfx 的点的 f（n）值里最小的为新的 maxfx，从头开始搜索。

```
morefx.append(x.fx)#maxfx 首先取起点的 f（n）
k = 0
while flag == 0:
    stack = []
    stack.append(x)
    k += 1
    maxfx = max(morefx)
    print(k, 'round', time.clock(), 's', 'maxfx=' + str(maxfx))
    morefx = []
    idastar(x, data, goaldis)
```

②IDA*单独一轮的搜索过程

由于迭代加深搜索本身的性质，找到的第一个解就是最优解。所以我在找到解后，直接 exit 退出，可以大大节省后面的搜索时间。同时，这里的不用存储结点的父节点，因为深度优先搜索的栈倒序输出即为搜索的路径。

```
def idastar(node1, nodedata, goaldis):
    global flag, maxfx
    if node1.fx > maxfx:
        morefx.append(node1.fx)#当前矩阵 f（x）大于 maxf（x）则回溯
        return
    if node1.currentdata == goal and flag == 0:#找到目标
        for item in stack:#输出深度搜索栈中的结点——形成路径
            if item.value != -1:
                print(item.value, end=' ')
        print('\n')
        print("step:", len(stack) - 1)
        flag = 1
        endtime = time.clock()
```

```
        print('time:', endtime - starttime)
        exit()#由深度优先搜索的性质，找到解即为最优解，退出
    for i in range(4):#空格进行上下左右移动
        newx = node1.x + behave[i][0]
        newy = node1.y + behave[i][1]
        newgx = node1.gx + 1#进行一次移动，gx+1
        if newx > 3 or newx < 0 or newy > 3 or newy < 0:
            continue
        newdata = copy.deepcopy(nodedata)
        tmp = nodedata[newx][newy]
        newdata[node1.x][node1.y] = tmp
        newdata[newx][newy] = 0
        newnode = node(tmp, newx, newy, newdata, newgx)
        newhx = 0
        for i in range(4):
            for j in range(4):
                if newdata[i][j] != 0:
                    a, b = goaldis[newdata[i][j]]
                    newhx += abs(i - a) + abs(j - b)#计算曼哈顿距离
        newnode.hx = newhx
        newnode.fx = newnode.hx + newnode.gx
        stack.append(newnode)#当前搜索的结点入栈
        idastar(newnode, newdata, goaldis)#继续深度搜索
        del (stack[-1])#回溯时删去该节点
```

实验结果及分析

实验数据

由于 python 本身的动态语言决定它是最慢的语言之一+电脑本身配置问题，我在完全写完两个程序之后才后悔没用 C++。（正如之前群里讨论的那样，A* 不存在的……）所以使用 PPT 里的数据跑 15 数码问题实在是花费太久时间了。但是 python 跑一些不是特别复杂的矩阵结构还是可以在较短时间内出结果的，尤其是使用了 pypy 解释器后，比 cpython 跑得快了几倍。由此，我设计了不同难度的三个矩阵进行对比。

首先是简单到飞起的手推矩阵。这个简单矩阵我打算用来做横向对比——即使用两个不同的 h（n）的效率对比。

1	2	3	4
5	6	7	8
9	11	4	0
13	10	15	12

接下来是两个复杂一些的矩阵，它们两个很接近，但是在 A*和 IDA*运行的效果差别比较大。我想用它们进行两个算法效率的纵向对比。

10	5	8	3
0	1	9	4
13	14	2	6
15	7	12	11

对这个矩阵增加几步：

10	5	8	3
1	9	2	4
13	14	0	6
15	7	12	11

实验结果展示

第一个简单矩阵的移动路径：

```
12 15 14 11 10 14 15

step: 7
```

第二个矩阵的移动路径：

```
10 5 1 9 8 3 4 8 2 6 11 12 7 15 13 10 9 2 6 7 15 14 10 9 5 1 2 6 7 11 12

step: 31
time: 0.5708229493466248
```

第三个矩阵的移动路径（此时相比于第二个矩阵加几步来说已经有了一个更优的路径）：

```
6 11 12 7 15 13 14 9 1 10 5 1 9 14 10 9 2 8 3 4 8 6 7 15 14 10 9 5 1 2 6 7 11 12

step: 34
```

评测指标展示（此处使用 pypy3 跑代码，可以比 cpython 跑快几倍）

1、 横向对比，使用不同的 h（n）

下图分别是一个 7 步的简单矩阵使用 ida*并不同的 h（n）的横向效果对比。其中第一个为“不在目标位置的数字个数”，第二个为曼哈顿距离。容易看出，使用“不在目标位置的数字个数”这个 h（n）速度较慢一些。

```
C:\Users\映雪\Desktop\AI实验>pypy3 idastar2.py
1 round 0.0001109331913388484 1 s maxfx=5
2 round 0.002346663662937178 s maxfx=7
12 15 14 11 10 14 15

step: 7
time: 0.06798881964097753

C:\Users\映雪\Desktop\AI实验>pypy3 idastar2.py
1 round 0.0002547196739588173 4 s maxfx=5
2 round 0.002280957080374937 3 s maxfx=7
12 15 14 11 10 14 15

step: 7
time: 0.015790913120964537
```

仔细分析也可以得知：每进行一次移动，曼哈顿距离变化的可能性大一些，而进行一次移动绝大多数情况下可能还是错误的位置。所以“不在目标位置的数字个数”这个 h（n）的收敛速度会比曼哈顿距离这个 h（n）收敛速度要慢很多。如果矩阵更加复杂一些，使用第一个 h（n）的话，收敛速度会特别慢，非常影响效率。（比如矩阵 2 我使用曼哈顿距离只需几秒，而不在目标位置上的数字个数就非常地慢）、

2、纵向对比，比较 A*和 IDA*

首先在 A*和 IDA*上分别运行矩阵 2，因为矩阵 2 的组成比较简单，所以两个都可以在短时间内跑出来：

```
C:\Users\映雪\Desktop\AI实验>pypy3 astar.py
10 5 1 9 8 3 4 8 2 6 11 12 7 15 13 10 9 2 6 7 15 14 10 9 5 1 2 6 7 11 12

step: 31
time: 0.33797716738922573

C:\Users\映雪\Desktop\AI实验>pypy3 idastar.py
1 round 0.0001937064187224507 s maxfx=31
10 5 1 9 8 3 4 8 2 6 11 12 7 15 13 10 9 2 6 7 15 14 10 9 5 1 2 6 7 11 12

step: 31
time: 0.1440962422234766
```

由上图可知，使用 ida*的时间比使用 a*的时间要快很多。

不过这是 A*短时间内能跑出来的，下面使用复杂的矩阵三。

```
C:\Users\映雪\Desktop\AI实验>pypy3 idastar.py
1 round 0.00011647985090579084 s maxfx=28
2 round 0.07431499821013562 s maxfx=30
3 round 0.2554578063473412 s maxfx=32
4 round 1.3180928195078576 s maxfx=34
6 11 12 7 15 13 14 9 1 10 5 1 9 14 10 9 2 8 3 4 8 6 7 15 14 10 9 5 1 2 6 7 11 12

step: 34
time: 4.1784343316040555

C:\Users\映雪\Desktop\AI实验>pypy3 astar.py
```

由图可知，ida*跑这个复杂的矩阵经历了 4 轮迭代加深，4 秒可以出结果，至于 a*……boom!

从设计 3 个不同复杂度的矩阵，进行纵向和横向的对比，我们可以得到下面两张表：

	不在目标位置的数字个数	**曼哈顿距离**
A*		效率更高
IDA*		效率更高

表 1：横向对比 h（n）

	A*	IDA*
时间		在本次实验中更少
空间		更少

表 2：纵向对比 A*和 IDA*