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	人工智能（陈川）	专业（方向）	计算机科学与技术人工智能与大数据
学号	20337025	姓名	崔璨明

一、实验题目

使用$A*$算法与$IDA*$算法解决15-Puzzle问题，启发函数可以自己选取，至少尝试两种不同的启发式函数，需要完成的基本样例和挑战样例如下：

基本样例：

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

挑战样例：

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

二、实验内容

1、算法原理

$A*$算法原理：

$A*$算法是一种启发性搜索算法，主要是用于在两点之间选择一个最优路径，主要通过一个估值函数(启发式函数)来引导搜索的方向，以得到最优的结果（可采纳的启发式函数低估了当前节点到达目标节点的成本，使得实际成本最小的最优路径能够被选上；如果启发函数大于这个上限， 则搜索算法出现发散，不能保证总能找到最优解），其算法步骤为：

1. 从起点开始，把其当成待处理点存入一个“open列表”。

2. 搜寻起点周围可能通过的节点，也把它们加入open列表，为这些节点计算 $f(x) = g(x) + h(x)$，并且将节点A存为“父节点”。
3. 从open列表中删除节点A，将其加入“close列表”(表示已经访问过)。循环直到找到目标节点或者“开启列表”为空。
4. 从“open列表”中找到估价函数值最低的节点C，并将它从“open列表”中删除，添加到“close列表”中。
5. 检查C所有相邻节点，将其加入“open列表”，将C作为它们的父节点。
6. 如果新的相邻节点已经在“open列表”，且f(x)值更小，则更新它们的f(x)值。

$IDA*$算法原理：

$IDA*$是迭代加深深度优先搜索算法（IDS）的扩展。因为它不需要去维护表，因此它的空间复杂度远远小于$A*$，在搜索图为稀疏有向图的时候，它的性能会比$A*$更好，$A*$索和宽度优先搜索或一致代价搜索一样存在潜在的空间复杂度过大的问题。$IDA*$ 迭代加深的A*搜索与迭代加深搜索一样用于解决空间复杂度的问题，就像迭代加深算法，但用于划定界限的不是深度，而是使用 f 值，在每次迭代时，划定的界限是 f 值超过上次迭代的界限最少的节点 的 f 值。可以证明，当启发函数h为可采纳时，IDA* 是最优的。

因此，选取一个合适的启发式函数对于提高搜索效率有着至关重要的作用，一个好的启发式函数应该具有可采纳性、一致性、最优性、单调性等等。

2、算法伪代码

通过$A*$算法解决15-puzzle问题的伪代码如下：

```
输入：15数码图的初始状态start
输出：移动的过程

frontier = PriorityQueue()     //建立一个优先队列，以f(x)的值升序排列
frontier.put(start)
came_from = dict()     //前驱节点
cost_so_far = dict()  // 花费的代价
came_from[start] =None
cost_so_far[start] =0

while frontier不为空:
        current = frontier.get()
         if current 为目标状态:
                 finish
                 打印路径...
         for next in neighours(current):    //当前状态可以到达的状态
                 new_cost = cost_so_far[current] + 1
         if next这个状态没有出现过，或代价更小:
                 cost_so_far[next] = new_cost
```

```
                priority = new_cost + h(x)     //h(x)为启发式函数
                frontier.put(next)
                came_from[next] = current      //记录前驱方便输出
```

通过$IDA*$算法解决15-puzzle问题的伪代码如下：

```
输入：15数码图的初始状态start
输出：移动的过程

void search(path, g, bound):
        f = g + h(node)
        if(f > bound):        return f
        if(node为最终状态):   return -1
 Min = MAX_NUM
 for succ in change(node):
        if succ not in path:
                path.append(succ)
                t = search(path, g+1, bound)
        if(t == -1):
                return -1
        if(t < Min):
                Min = t
                path.pop() //此路不通，删除记录
        return Min

void ida_star(start):
        bound = h(start)
        path = [start]
        while(True):
                t = search(path, 0, bound)
                if(t == -1):
                        return (path, bound)
                if(t > 70):     //这里我设置了70为限定值
                        return ([], bound)
        bound = t

while(1):
        bound=1
        ida_star(start)
        if 找到解:
                print...
                break
        else:
```

```
                bound++
```

3、关键代码展示

在选取启发式函数上，我采用了三种不同的启发式函数进行对比，一种是以曼哈顿距离作为$h(x)$的值，一种是以错牌数来作为$h(x)$，还有一种是以(错牌数 $*0.5+$ 曼哈顿距离 $*0.*5$)的值来作为$h(x)$，在相同的程序下进行测试，测试程序为python实现的$A*$搜索。这三种启发式函数的关键代码如下：

曼哈顿距离：

```
#计算曼哈顿距离
def h(node):
    cost = 0
    for i in range(4):
        for j in range(4):
            num = node[i][j]
            x,y = dis[num]
            cost += abs(x-i) + abs(y-j)
    return cost
```

错牌数：

```
def h2(node):
    cost=0
    for i in range(4):
        for j in range(4):
            if node[i][j]!=i/4+j+1:
                cost+=1
    return cost
```

两种加权：

```
def h3(node):
    cost=0
    cot=0
    for i in range(4):
        for j in range(4):
            num = node[i][j]
            x,y = dis[num]
            cost += abs(x-i) + abs(y-j)
            if num!=i/4+j+1:
                cot+=1
    return cost*0.5+cot*0.5
```

以python程序为例，$A*$算法执行过程的关键代码如下：

```python
while not frontier.empty():
    current = frontier.get()[1]
    print(current)
    if current == goal:
        break
    for next in neighours(current):
        #new_cost = cost_so_far[current] + graph.cost(current, next)
        new_cost = cost_so_far[current] + 1
        if next not in cost_so_far or new_cost < cost_so_far[next]: #没有走过或代价更小
            cost_so_far[next] = new_cost
            #priority = new_cost + heuristic(next)
            priority = 0.5*new_cost + heuristic(next)
            frontier.put( (priority,next) )
            came_from[next] = current

ans=[]
k=goal
while k!=None:
    ans.append(k)
    k=came_from[k]
```

以python程序为例，$IDA*$算法执行过程关键代码如下：

```python
def search(path, g, bound):
    node = path[-1]     #选取最小f值的状态
    f = g + h(node)     #计算f值
    if(f > bound):      #大于阈值则返回
        return f
    if(is_cmp(node)):   #如果等于目标状态则结束
        return -1
    Min = 999999
    for succ in change(node):   #生成下一个状态
        if succ not in path:
            path.append(succ)
            t = search(path, g+1, bound)  #递归深搜
            if(t == -1):
                return -1
            if(t < Min):
                Min = t
            path.pop()      #此路不通删除记录
    return Min
```

由于python为解释性语言，因此相同的算法过程python的运行时间要比C++长很多，所以我为了方便对两种算法进行对比，运用C++对算法进行了实现，和以C++程序为例，$A*$算法执行过程关键代码如下：

```cpp
    frontier.push(start);//优先队列
    close[start.sid]=start.f;//close表

    clock_t t_start,end;
    t_start=clock();//计时
    while(!frontier.empty()){
        state current = frontier.top();
        //取f值最小的
        if(current.g>40)
            break;
        cout<<current.g<<endl;
        if(is_goal(current)){
            cout<<"steps:"<<current.g-1<<endl;
            break;//找到了便进行输出
        }
        frontier.pop();
        vector<state> next_states=neibour(current);//找到邻接的状态

        for(int i=0;i<next_states.size();i++){

            if(close.find(next_states[i].sid) == close.end() || close[next_states[i].sid]>next_states[i].f ){
                close[next_states[i].sid]=next_states[i].f;//如果没出现过或者f值更小 ， 入队
                frontier.push(next_states[i]);
            }
        }
```

以C++程序为例，$IDA*$算法执行过程关键代码如下：

```
void dfs(int x,int y,int len,int pre_move)
{
    if(flag) return;
    int dist=h_manhadun(maze);
    if(len==bound)
    {
        if(dist==0)  //成功 退出
        {
            flag=1;
            lens=len;
            return;
        }
        else {
            if(!ans.empty())
            ans.pop_back();
            return;
        } //超过预设长度 回退
    }
    for(int i=0;i<4;i++)
    {
        if(i+pre_move==3&&len>0)
        continue;//不移动回去
        int tx=x+ moves[i][0];
        int ty=y+ moves[i][1];
        if(tx>=0&&tx<4&&ty>=0&&ty<4)
        {
            swap(maze[x][y],maze[tx][ty]);

            ans.push_back(maze[x][y]);

            int p=h_manhadun(maze);
            if(p+len<=bound&&flag==0)
            {
```

三、实验结果

六个样例的解答：

首先是对给出的6个案例的解决方案（目前我编写的程序中，只有ida算法的程序能在较快的时间内给出解），为了提高程序运行时间和减小内存开支，我设置输出序列为和数码0依次交换的数字序列：

测试样例 1：

```
PS E:\VSCODE\CPP> cd "e:\VSCODE\CPP\" ; if ($?) { g++ extract.cpp -o extract } ; if ($?) { .\extract }
1 2 4 8
5 7 11 10
13 15 0 3
14 6 9 12
steps:22
using time:0ms
swap orders:
12-11-10-9-13-14-9-6-7-3-4-8-3-10-15-9-6-15-11-3-10-11
PS E:\VSCODE\CPP>
```

可以看到，对于移动步骤为22的15数码的解，程序能在1ms内给出解。

测试样例 2：

```
PS E:\VSCODE\CPP> cd "e:\VSCODE\CPP\" ; if ($?) { g++ extract.cpp -o extract } ; if ($?) { .\extract }
5 1 3 4
2 7 8 12
9 6 11 15
0 13 10 14
steps:15
using time:1ms
swap orders:
15-14-10-6-2-1-5-2-7-8-12-15-14-10-13
PS E:\VSCODE\CPP>
```

对于移动步骤为15的问题，程序也能够很快给出解决方案。

测试样例 3：

```
PS E:\VSCODE\CPP> cd "e:\VSCODE\CPP\" ; if ($?) { g++ extract.cpp -o extract } ; if ($?) { .\extract }
14 10 6 0
4 9 1 8
2 3 5 11
12 13 7 15
steps:49
using time:2981ms
swap orders:
15-11-7-6-10-7-12-8-6-10-5-9-13-14-7-12-11-7-12-13-14-12-13-5-2-3-4-6-8-11-5-2-3-1-9-14-2-3-1-4-10-1-4-9-14-4-9-10-6
PS E:\VSCODE\CPP>
```

可以看到，对于测试样例3，需要2981ms。即2.981s。

测试样例 4：

```
PS E:\VSCODE\CPP> cd "e:\VSCODE\CPP\" ; if ($?) { g++ extract.cpp -o extract } ; if ($?) { .\extract }
6 10 3 15
14 8 7 11
5 1 0 2
13 12 9 4
steps:48
using time:12090ms
swap orders:
12-11-7-8-4-3-2-6-10-14-13-9-5-1-6-10-14-7-15-13-9-5-1-14-8-15-11-4-15-2-3-15-2-11-7-9-13-12-4-2-11-7-9-1-5-13-12-9
PS E:\VSCODE\CPP>
```

对于48步的测试样例4，需要12090ms，即12s。

测试样例 5（挑战样例）：

```
PS E:\VSCODE\CPP> cd "e:\VSCODE\CPP\" ; if ($?) { g++ extract.cpp -o extract } ; if ($?) { .\extract }
11 3 1 7
4 6 8 2
15 9 10 13
14 12 5 0
steps:56
using time:558455ms
swap orders:
12-8-7-3-2-6-3-11-10-9-5-3-11-2-4-7-8-10-15-12-10-8-2-4-6-1-3-11-9-14-13-5-14-15-12-13-5-14-15-12-8-10-13-5-12-9-4-6-1-3-11-4-6-8-10-13
PS E:\VSCODE\CPP>
```

测试样例5的步数为56，难度系数大大增加，因此程序的运行时间也增加到了558455ms，即558.5s，需要差不多十分钟的时间。

测试样例 6（挑战样例）：

```
PS E:\VSCODE\CPP> cd "e:\VSCODE\CPP\" ; if ($?) { g++ extract.cpp -o extract } ; if ($?) { .\extract }
0 5 15 14
7 9 6 13
1 2 12 10
8 11 4 3
steps:62
using time:675709ms
swap orders:
15-14-13-9-10-11-14-15-12-8-4-3-7-14-15-13-11-10-5-1-2-7-14-15-13-12-8-4-3-14-15-13-12-8-4-3-13-12-10-11-8-4-3-10-12-6-1-5-9-8-11-1-5-2-7-5-2-9-1-2-9-7
PS E:\VSCODE\CPP>
```

测试样例6的步骤是最多的，需要经过62步的移动，因此算法花费的时间为675709ms，需要11分钟的时间来找到最优路径。

四、两种算法对比

为了将$A*$算法和$IDA*$算法进行对比，需要控制变量，因此我选取了python版本实现的$A*$和$IDA*$算法进行对比，并且两个程序对状态的储存方式等与算法运行无关的部分尽量控制相同，最后通过`time`库和`os`库中的函数显示出程序运行的时间和占用的内存空间，对于同一个**测试样例1**，两者运行结果如下：

$A*$算法：

```
------------------------
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0
------------------------
23
using time: 0.11401081085205078
当前进程的内存使用：0.0267 GB
PS E:\VSCODE\py>
```

$IDA*$算法：

```
[13, 14, 15, 12]
steps 22
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 0]
bound 26
using time: 0.013998985290527344
当前进程的内存使用：0.0135 GB
PS E:\VSCODE\py>
```

对于**测试样例2**，两者结果如下：

$A*$算法：

```
------------------------
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0
------------------------
16
using time: 0.04300522804260254
当前进程的内存使用：0.0266 GB
PS E:\VSCODE\py>
```

$IDA*$算法：

```
[13, 14, 0, 15]
steps 15
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 0]
bound 21
using time: 0.0029969215393066406
当前进程的内存使用：0.0135 GB
```

测试样例	$A*$	用时	内存开销	$IDA*$	用时	内存开销
1		0.11401ms	0.0267GB		0.014ms	0.0135GB
2		0.043ms	0.0266GB		0.003ms	0.0135GB

如结果显示，在解决测试样例1的程序时，使用$A*$算法的python程序花费的时间为0.11401ms，使用的内存为0.0267GB，而使用$IDA*$算法的python程序花费的时间为0.014ms，花费的内存为0.0135GB。

在解决测试样例2的程序时，使用$A*$算法的python程序花费的时间为0.043ms，使用的内存为0.0266GB，而使用$IDA*$算法的python程序花费的时间为0.003ms，花费的内存为0.0135GB。

由对比可知，无论是时间花费还是内存开销，使用$IDA*$算法的python程序都比使用$A*$算法的python程序要好，但这也有可能是我编写程序时的问题，例如在编写$A*$算法时保存状态的各种函数和操作可能花费了太多时间，相比起不需要保存中间状态的$IDA*$就显得不利。需要再进一步查找资料进行更详细的对比。

通过进一步的对比和实例分析，得出两种算法各自的优缺点如下：

算法	优点	缺点
$A*$算法	与广度优先搜索策略和深度优先搜索策略相比，算法不是盲目搜索，而是有提示的搜索，搜索过程中使得h(n)逐渐接近h*(n)，最终找到最优路径。	一般要使用大量的空间用于存储已搜索过的中间状态，防止重复搜索
$IDA*$算法	迭代加深搜索算法，在搜索过程中采用估值函数，以减少不必要的搜索。使用回溯方法，不用保存中间状态，相较于$A*$大大节省了空间	导致重复搜索，回溯过程中每次depth变大都要再次从头搜索。在某些情况下使用时间要长。

五、三种启发式函数性能对比

在实验中，我编写了三种启发式函数，分别是以曼哈顿距离作为$h1(x)$的值，一种是以错牌数来作为$h2(x)$，还有一种是以$(\text{错牌数} * 0.5 + \text{曼哈顿距离} * 0.*5)$的值来作为$h3(x)$，在相同的程序（ida_star.cpp）下进行测试，选取测试样例1，测试结果如下：

h1曼哈顿距离：

```
PS E:\VSCODE\CPP> cd "e:\VSCODE\CPP\" ; if ($?) { g++ extract.cpp -o extract } ; if ($?) { .\extract }
1 2 4 8
5 7 11 10
13 15 0 3
14 6 9 12
steps:22
using time(h1):0ms
swap orders:
12-11-10-9-13-14-9-6-7-3-4-8-3-10-15-9-6-15-11-3-10-11
PS E:\VSCODE\CPP>
```

h2错牌数：

```
PS E:\VSCODE\CPP> cd "e:\VSCODE\CPP\" ; if ($?) { g++ extract.cpp -o extract } ; if ($?) { .\extract }
1 2 4 8
5 7 11 10
13 15 0 3
14 6 9 12
steps:22
using time(h2):13ms
swap orders:
12-11-10-9-13-14-9-6-7-3-4-8-3-10-15-9-6-15-11-3-10-11
PS E:\VSCODE\CPP>
```

h3 $0.5h1 + 0.5h2$**:**

```
PS E:\VSCODE\CPP> cd "e:\VSCODE\CPP\" ; if ($?) { g++ extract.cpp -o extract } ; if ($?) { .\extract }
1 2 4 8
5 7 11 10
13 15 0 3
14 6 9 12
steps:22
using time(h3):3ms
swap orders:
12-11-10-9-13-14-9-6-7-3-4-8-3-10-15-9-6-15-11-3-10-11
PS E:\VSCODE\CPP>
```

启发式函数	运行时间	输出结果
曼哈顿距离总和	<1ms	正确
错牌数	13ms	正确
错牌数0.5+曼哈顿距离* 0.5	3ms	正确

由对比可知，选取启发式函数为h1(曼哈顿距离总和)时，算法的效率最高，选取错牌数为启发式函数时，算法的效率最低。这是因为相比起其他两个，曼哈顿距离作为启发式函数时，估计值和真实值的差距较小，这会大大影响算法的性能。

估计值和真实值的差距会造成算法性能差距的原因:

由于启发式函数是当前状态到目标状态的距离的一个估计，有引导算法前进步骤的作用，因此可以用来控制$A*$的行为，这也是不同启发式函数造成算法性能差距的原因，当选取的估计值距离真实值的差距较大（如使用错牌数作为启发式函数），由于将一个数码移动到相应的位置可能需要好几步，且移到后还可能需要不断移动，因此真实代价会远远大于错牌数（错牌数将代价看错1），虽然曼哈顿距离也李真实值有一定的差距，但和错牌数相比，其估计值距离真实值差距较小，因此可以更加正确地引导算法的前进，往接近目标状态的方向移动，这也导致了两者的运行时间的差距。

六、创新点&优化

在一开始编写的用python实现的$A*$算法程序中，我简单地运用了二维列表来存储每个状态，但这却给程序的运行带来了极大的负担，当搜索的深度加大时，二维列表产生的开销会非常大，这也导致了我一开始写的程序执行算法速度极慢，对步数较大的测例可能需要几个小时来计算，但后来经过查找资料、和同学交流，我将存储结构由二维列表改成了一维的元组，虽然

元组的操作较列表麻烦，但这种改进大大提高了程序的运行效率，减少了算法的运行时间，这是因为：

1. 元组缓存于Python运行时环境，这意味着我们每次使用元组时无须访问内核去分配内存。
2. 元组可以被轻松快速地创建，因为它们可以避免跟操作系统频繁的打交道，而后者会花很长的时间

七、实验总结和感想

通过这次实验，我对两种启发式算法——$A*$算法和$IDA*$算法有了更加深刻的理解，且能独立实现相应的程序，用以解决15数码问题，这对我的能力有了很大的提升，除此之外，通过对算法的对比和对启发函数的对比，我学到了很多提高算法效率的知识，这也让我受益匪浅。

八、参考资料

- 人工智能第七讲 正式版.pdf