Machine Learning Homework 1.1

专业：软件工程　　姓名：沈金龙　　学号：18214806

1. 实验题目

8-puzzle problem

Given any randomly generated start state and a goal state shown below, implement the IDS, greedy search and A* search algorithms, respectively, to find a sequence of actions that will transform the state from the start state to the goal state.

2. 实验要求

1) When implementing the A* search algorithm, you need to use at least two different heuristic functions.

2) Compare the running time of these different searching algorithms, and do some analyses on your obtained results. Note that the reported running time should be averaged on many randomly generate different start states.

3. 实验过程及代码

1）　本实验采用 python3.6 完成。

2）　问题的搜索形式描述：

状态：状态描述了 8 个棋子和空位在棋盘的 9 个方格上的分布。

初始状态：任何状态都可以被指定为初始状态。

操作符：用来产生 4 个行动（上下左右移动）。

目标测试：用来检测状态是否能匹配上图的目标布局。

路径费用函数：每一步的费用为 1，因此整个路径的费用是路径中的步数。

现在任意给定一个初始状态，要求找到一种搜索策略，用尽可能少的步数得到指定的目标状态。

3） 启发式搜索：

启发式搜索(Heuristically Search)又称为有信息搜索(Informed Search)，它是利用问题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，这种利用启发信息的搜索过程称为启发式搜索。

启发式搜索包括 A 算法和 A*算法。

启发式算法的核心思想：f(x)=g(x)+h(x)

评估函数 f(x)定义为：从初始节点 S0 出发，约束地经过节点 X 到达目标节点 Sg 的所有路径中最小路径代价的估计值。

其一般形式为 f(x)=g(x)+h(x)，g(x)表示从初始节点 S0 到节点 X 的实际代价；h(x)表示从 X 到目标节点 Sg 的最优路径的估计代价。

4） 如何判断数码问题有解？

八数码问题的一个状态实际上是 0~8 的一个排列，对于任意给定的初始状态和目标，不一定有解，也就是说从初始状态不一定能到达目标状态。

因为排列有奇排列和偶排列两类，从奇排列不能转化成偶排列，从偶排列也不能转化成奇排列。因此，可以在运行程序前检查初始状态和目标状态的窘是否相同，相同则问题可解，应当能搜索到路径。否则无解。

一个状态表示成一维的形式，求出除 0 之外所有数字的逆序数之和，也就是每个数字前面比它大的数字的个数的和，称为这个状态的逆序。用 F(X)表示数字 X 前面比它大的数的个数，全部数字的 F(X)之和为 Y=∑(F(X))，如果 Y 为奇数则称原数字的排列是奇排列，如果 Y 为偶数则称原数字的排列是偶排列。

若两个状态的逆序奇偶性相同，则可相互到达，否则不可相互到达。由于原始状态的逆序为 0（偶数），则逆序为偶数的状态有解。也就是说，逆序的奇偶将所有的状态分为了两个等价类，同一个等价类中的状态都可相互到达。

5） 四种常见的解决算法：

(1) 宽度优先搜索算法：

如果搜索是以接近起始节点的程度依次扩展节点的，那么这种搜索就叫做宽度优先搜索。这种搜索是逐层进行的，在对下一层的任一节点进行搜索之前，必须搜索完本层的所有节点。

(2) 有序搜索(A 算法) ：

令 f(n)表示节点 n 的估价函数值，估算节点希望程度的量度。本次实验选择的 f(n)的函数形式为： f(n)=g(n)+h(n)，其中, g(n)为初始节点到当前节点的路径长度(深度), h(n)为当前节点“不在位”的将牌数。

有序搜索（ordered search），即最好优先搜索, 选择 Open 表上具有最小 f 值的节点作为下一个要扩展的节点。

(3) A*算法：

A*算法是由 f(x)=g*(x)+h*(x)决定，在 A 算法的基础上添加了约束条件，g*(x)，h*(x)<=任意 h(x)；A*算法是最优的 A 算法（因为估值函数最优）。

(4) 深度优先搜索算法：

首先扩展最新产生的(即最深的)节点。防止搜索过程沿着无益的路径扩展下去，往往给出一个节点扩展的最大深度（深度界限）。与宽度优先搜索算法最根本的不同在于：将扩展的后继节点放在 OPEN 表的前端。

6） 核心代码展示：

(1) 广度优先搜索算法实现：

```
def breadthFirstSearch(self):  # 广度优先搜索
    if(self.root.puzzled.check() == False):
        print("there is no way to get to the goal state!")
        return
    numTree = NumTree()
    numTree.insert(self.root.puzzled.toOneDimen())
    t = [self.root]
    flag = True
    generation = 0
    while(flag):
        # print("it's the "+str(generation) + " genneration now,the total num of items is "+str(len(t)))
        tb = []
        for i in t:

            if(i.puzzled.isRight() == True):
                # i.displayToRootNode()
                flag = False
                break
            else:
                for j in i.puzzled.getAbleMove():
                    tt = i.puzzled.clone()
                    tt.move(j)
                    a = node(tt)
                    if(numTree.searchAndInsert(a.puzzled.toOneDimen()) == False):
                        i.addChild(a)
                        tb.append(a)
        t = tb
        generation += 1
```

(2) 深度优先搜索算法实现：

```
def depthFirstSearch(self):  # 深度优先搜索
    if(self.root.puzzled.check() == False):
        print("there is no way to get to the goal state!")
        return
    numTree = NumTree()
    numTree.insert(self.root.puzzled.toOneDimen())
    t = self.root
    flag = True
    gen = 0  # 深度有界，限制为6层，超过不再扩展
    while(flag):
        # print("genneration: "+str(gen))
        if(gen == 6):
            break
        if(t.puzzled.isRight() == True):
            # t.displayToRootNode()
            flag = False
            break
        else:
            f1 = True
            for j in t.puzzled.getAbleMove():
                tt = t.puzzled.clone()
                tt.move(j)
                a = node(tt)
                if(numTree.searchAndInsert(a.puzzled.toOneDimen()) == False):
                    t.addChild(a)
                    t = a
                    f1 = False
                    gen += 1
                    break
            if(f1 == True):
                t = t.father
                gen -= 1
```

(3) A*搜索算法实现：

```
def AStarSearch(self):  # A*
    if(self.root.puzzled.check() == False):
        print("there is no way to get to the goal state!")
        return
    numTree = NumTree()
    numTree.insert(self.root.puzzled.toOneDimen())
    leaves = [self.root]
    leavesFn = [0]
    while True:
        t = leaves.pop()  # open表
        # print(leavesFn.pop())
        if(t.puzzled.isRight() == True):
            # t.displayToRootNode()
            break
        for i in t.puzzled.getAbleMove():
            tt = t.puzzled.clone()
            tt.move(i)
            a = node(tt)
            if(numTree.searchAndInsert(a.puzzled.toOneDimen()) == False):  # close表
                t.addChild(a)
                fnS = self.__sortInsert(leavesFn, a.getFn())
                leaves.insert(fnS, a)
```

4. 实验结果与分析

1） 三种算法的运行时间比较，并取三十次有效状态的平均值，如下图所示：

序号	广度优先搜索算法	深度优先搜索算法	A*搜索算法
1	56.59ms	8.00ms	13.03ms
2	8.02ms	7.00ms	4.01ms
3	42.55ms	7.00ms	13.02ms
4	23.03m	8.00ms	8.02ms
5	8.99ms	8.01ms	4.00ms

本次实验采用自动生成随机初始状态的方式，共计实验了 30 组，上图仅取前五组数据进行展示，全部实验的平均数据为：

Avgtime：bfs 11.03ms - dfs 5.37ms - astar 4.86ms

5. 总结

本实验回顾了广度优先搜算法、深度优先搜索算法以及基于启发式搜索的 A*算法，将其应用到解决八数码问题，并完成基于 python 的实现。在实验中发现，基于启发式搜索的 A*算法的效率最高，原因是未将所有的分支节点当成同等代价，而是由估价函数进行节点过滤，极大的加快了算法的运行，方便找到最终的状态。