实验报告

——A*算法

姓名：蔡梓珩　　学号：16340008　　日期：2018/12/18

摘要：利用 A*算法解决八数码问题，比较不同启发函数（h1，h2）的搜索效率，并验证关于 A*算法的命题。

1．导言

本实验主要使用 A*算法解决八数码问题。

八数码问题主要是由 8 个 1-8 的数字以及一个空格组成一个九宫格，通过移动空格若干次使得九宫格中数字到达以下目标状态：

1	2	3
8		4
7	6	5

对于每个状态，可以往不同方向移动空格使得该状态产生多个不同的新状态作为后继节点，于是对于候选状态的选择策略可以有 BFS、DFS、启发式搜索等。对于八数码问题，我们可以理解成一种路径搜索问题，以其实状态为起点，目标状态为终点，进行搜索。每个节点的可搜索路径为其空格往四个（或更少）方向移动后得到的、不属于其前驱节点的状态。

2．实验过程

A 算法即对于候选节点每个节点 n，有 f(n)，根据 f(n)排序并选择最优者。其中 f(n)定义为

$$f(\mathrm{n}) = g(n) + h(n)$$

其中 g(n)为起始状态到 n 的距离，h(n)为 n 到目标状态的预估距离。

若进一步规定 h(n)≥0，并且定义：

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

其中，f*(n)表示起点经过 n 到终点的最优路径搜索费；g*(n)表示起点到 n 的实际最小费用；h*(n)表示 n 到终点的实际最小费用的估计。

当要求 h(n)≤h*(n)，就称这种 A 算法为 A*算法。

A*算法解决八数码问题的主要流程如下：

设 S_0 为起始状态，S_g 为目标状态：

1. Open={ S_0}
2. Closed={}
3. 如果 Open={}，失败退出
4. 在 Open 表中取出其末尾元素 n，n 放到 Closed 表中
5. 若 n∈S_g，则成功退出。
6. 产生 n 的一切非 n 前驱节点的后继，并构成集合 M。
7. 对 M 中的元素 P，分别作两类处理：

a) 若 P∉G（G=Open+Closed），说明 P 并未被搜索到过，则将其计算 f(n),并将其插入 Open 表的合适位置，维护 Open 表从大到小的顺序，并将其加入 G 表中。（这里 f(n)=g(n)+h(n)，根据不同 h(n)有不同的启发式函数，不同的搜索策略）

b) 若 P∈G，说明 P 已经是别的节点的后序节点，则比较从 n 到 P 以及从 P.father 到 P 的消耗，并决定是否更改 P.father

8. 转第三步

在我的代码中，用一个类保存一个状态，每个状态包括一个节点的九宫格信息以及其父节点的指针。即，Tree 的信息跟每个节点的状态存储在一个对象中。

在算法退出后，保存最后得到的目标状态，一路根据其 node.father 追溯到根节点（起始状态），再逆序，即可得到路径。

使用的 f(n)中 h1(n)为九宫格中格子与数字不对应的个数，h2(n)为九宫格中每个格子中数字到正确位置需要消耗（横向纵向移动次数总和），即曼哈顿距离之和。比较两者 Open 表长度和到达目标的步数。

3．结果分析

运行环境为 windows10，Intel Core i5-8400 2.81GHz，RAM 2667MHz 16GB

编译器 PyCharm，语言 Python3

利用 A*算法求解八数码问题，在输出界面上动态显示 OPEN 表的结点数和评估函值最小的结点

比较两种启发函数（上课和书上讲到的 h1(n)和 h2(n)）的搜索效率，在输出界面上动态显示 OPEN 表的结点数、总扩展的结点数和评估函值最小的结点

输出 OPEN 表中在最佳路径上的结点及其评估函数值。

先使用课本上的样例进行测试：

2	8	3
1	6	4
7		5

迭代次数	h1(n)	h2(n)
1	Step: 1 Len of Open: 1 Len of G: 1 The best node: [2, 8, 3] [1, 6, 4] [7, 0, 5] The f(n) of this node: 4	Step: 1 Len of Open: 1 Len of G: 1 The best node: [2, 8, 3] [1, 6, 4] [7, 0, 5] The f(n) of this node: 5

2	Step: 2 Len of Open: 3 Len of G: 4 The best node: [2, 8, 3] [1, 0, 4] [7, 6, 5] The f(n) of this node: 4	Step: 2 Len of Open: 3 Len of G: 4 The best node: [2, 8, 3] [1, 0, 4] [7, 6, 5] The f(n) of this node: 5
3	Step: 3 Len of Open: 5 Len of G: 7 The best node: [2, 0, 3] [1, 8, 4] [7, 6, 5] The f(n) of this node: 5	Step: 3 Len of Open: 5 Len of G: 7 The best node: [2, 0, 3] [1, 8, 4] [7, 6, 5] The f(n) of this node: 5
4	Step: 4 Len of Open: 6 Len of G: 9 The best node: [2, 8, 3] [0, 1, 4] [7, 6, 5] The f(n) of this node: 5	Step: 4 Len of Open: 6 Len of G: 9 The best node: [0, 2, 3] [1, 8, 4] [7, 6, 5] The f(n) of this node: 5
5	Step: 5 Len of Open: 7 Len of G: 11 The best node: [0, 2, 3] [1, 8, 4] [7, 6, 5] The f(n) of this node: 5	Step: 5 Len of Open: 6 Len of G: 10 The best node: [1, 2, 3] [0, 8, 4] [7, 6, 5] The f(n) of this node: 5
6	Step: 6 Len of Open: 7 Len of G: 12 The best node: [1, 2, 3] [0, 8, 4] [7, 6, 5] The f(n) of this node: 5	Step: 6 Len of Open: 7 Len of G: 12 The best node: [1, 2, 3] [8, 0, 4] [7, 6, 5] The f(n) of this node: 5

7	Step: 7 Len of Open: 8 Len of G: 14 The best node: [1, 2, 3] [8, 0, 4] [7, 6, 5] The f(n) of this node: 5	结束
路径	[[2, 8, 3], [1, 6, 4], [7, 0, 5]] f(n): 4 [[2, 8, 3], [1, 0, 4], [7, 6, 5]] f(n): 4 [[2, 0, 3], [1, 8, 4], [7, 6, 5]] f(n): 5 [[0, 2, 3], [1, 8, 4], [7, 6, 5]] f(n): 5 [[1, 2, 3], [0, 8, 4], [7, 6, 5]] f(n): 5 [[1, 2, 3], [8, 0, 4], [7, 6, 5]] f(n): 5	[[2, 8, 3], [1, 6, 4], [7, 0, 5]] f(n): 5 [[2, 8, 3], [1, 0, 4], [7, 6, 5]] f(n): 5 [[2, 0, 3], [1, 8, 4], [7, 6, 5]] f(n): 5 [[0, 2, 3], [1, 8, 4], [7, 6, 5]] f(n): 5 [[1, 2, 3], [0, 8, 4], [7, 6, 5]] f(n): 5 [[1, 2, 3], [8, 0, 4], [7, 6, 5]] f(n): 5

观察上表可以看到，h2(n)可以迭代更少次数，而且 Open 表和 G 表讲更短。

随机选取了 5 个可行测例，比较 h1(n)与 h2(n)的效率：

测例	h1(n)（迭代数/耗时）	h2(n)（迭代数/耗时）
[1, 2, 4] [3, 8, 6] [5, 0, 7]	Step: 6034 Cost: 9.5 s	Step: 928 Cost: 0.30 s
[0, 3, 8] [7, 4, 6] [2, 1, 5]	Step: 3126 Cost: 2.3 s	Step: 240 Cost: 0.03 s
[7, 6, 3] [8, 2, 1] [0, 5, 4]	Step: 3352 Cost: 2.5 s	Step: 489 Cost: 0.09 s
[0, 7, 8] [5, 1, 3] [2, 4, 6]	Step: 2968 Cost: 1.9 s	Step: 150 Cost: 0.02 s
[6, 3, 2] [7, 5, 8] [1, 0, 4]	Step: 5902 Cost: 8.0 s	Step: 611 Cost: 0.13 s
[2, 7, 0] [8, 4, 1] [3, 5, 6]	Step: 38166 Cost: 322.7 s	Step: 3730 Cost: 3.37 s

可以很清晰比较到两者的差距，具有更多信息的 h2 的将更快搜索到目标。

验证凡 A*算法挑选出来求后继的点 n 必定满足: f(n）≤f*(S0)：

f*(S0)表示实际最小费用，用 `print("f*(S0):", len(path)-1)` 路径长-1 表示。（len(path)表示路径经过的点，减 1 后则为路径边的个数）

随机三个可行测例进行三个测试（使用 h2(n)）

	f*(S0)	f(n)范围
[5, 6, 8], [3, 0, 4], [7, 1, 2]	24	[18, 24]

[5, 7, 6], [0, 2, 4], [1, 8, 3]	25	[17, 25]
[2, 4, 6], [7, 3, 1], [0, 8, 5]	24	[14, 24]

验证 h1(n)的单调性，显示凡 A*算法挑选出来求后继的点 ni 扩展的一个子结点 nj，检查是否满足: h(ni）≤1+h(nj):

```
h(ni): 4 h(nj)+1: 4 h(nj)+1: 6 h(nj)+1: 6
h(ni): 3 h(nj)+1: 4 h(nj)+1: 5 h(nj)+1: 4
h(ni): 3 h(nj)+1: 5 h(nj)+1: 3
h(ni): 3 h(nj)+1: 4 h(nj)+1: 5
h(ni): 2 h(nj)+1: 2
h(ni): 1 h(nj)+1: 1 h(nj)+1: 3
```

这里使用了课本上的测例，随机测例由于步数过多无法写进报告。因此对于随机测例，我在添加了如下代码：

```
if not n.h1 <= P.h1 + 1:
    print("!!!!!!!!!!!!")
```

对于每个扩展的 P（即 nj）作判断，若有违反命题的则打印。结果没有打印。

如果将空格看作 0，即九数码问题，利用相似的启发函数 h1(n)和 h2(n)，求解相同的问题的搜索图是否相同？

不一样。

```
3 8 2
1 4 6
7 0 5
```
当初始起点如左图

把空格看作 0，h1(n)需要迭代步数 6882，h2(n)需要迭代步数 1352

而空格不算入 h1(n)和 h2(n)的话，h1(n)需要迭代步数 6507，h2(n)需要迭代步数 1246

每次迭代都要取节点并扩展，因此搜索图将是不一样的。

同时它也不是一个 A*算法。假设从目标状态回退一步（即 0 往任意方向与邻居交换），得到的状态的 h1(n)与 h2(n)都是 2，但实际其与目标的距离都是 1，违反了 h(n)<=h*(n)的规则。

写出能否达到目标状态的判断方法

将 3*3 的格子以行为顺序排列成一行数组 A。

对于 A 中每个数字 n，定义 f(n)为 A 中数字 n 前面比 n 小的数字的个数。定义 SUM 为 f(1)到 f(8)的总和。

我们可以验证 SUM 的奇偶性将一直保持不变：

1. 当空格左右移动，所有 f(n)都保持不变，SUM 不变
2. 当空格向上移动（设与 n 交换），则在数组 A 中介于空格和 n 中的数有 2 个，记为 a, b。
   a) a < n and b < n，则 f(n) += 2，SUM += 2
   b) a < n and b > n，则 f(n) += 1，f(b) -= 1，SUM 不变
   c) a > n and b > n，则 f(a) -= 1，f(b) -= 1，SUM -= 2
3. 当空格向下移动，同理

可以发现 SUM 永远以 2 为单位加减变化，其奇偶性不变，因此

起点的奇偶性与终点的奇偶性需一样，这是可达的必要条件。
然而其充分性，我没有能力证明。

实际上应该使用 numpy 等提升数组操作的速率，然而发现这点时代码已经成型不方便改动。或者使用 java，因为该项目不太需要作图。

4．结论

A*算法能更智能的选择搜索路径，而摆脱盲目搜索。A*算法将考虑每个候选节点的属性（已耗费代价），并预估其可能耗费的代价（小于实际消耗）。当预估的代价越接近实际消耗，说明其考虑更多因素，在选择时拥有更多信息，每次选择更接近正确路径的节点，减少迭代次数和待搜索对象。因此在使用 A*算法时考虑好的预估策略十分重要，如 h2 与 h1，同样作为 A*算法，消耗有近百倍的差距。

主要参考文献(三五个即可)

八数码问题有解的条件及其推广 https://blog.csdn.net/tiaotiaoyly/article/details/2008233