**实验一 搜索策略**

1. **实验内容**
2. 熟悉和掌握启发式搜索的定义、估价函数和算法过程，比较不同算法的性能。
3. 以八数码问题或路径规划等问题为例设计启发式搜索算法，改变启发函数，观察结果的变化，分析原因。

**二、 实验目的**

熟悉和掌握各种启发式搜索策略的思想，掌握A\*算法的定义、估价函数和算法过程，理解求解流程和搜索顺序。

**三、 实验内容**

1、分别以各种搜索算法为例演示搜索过程，比较不同算法的性能；

2、分析各种算法中的OPEN表CLOSE表的生成过程；

3、分析估价函数对搜索算法的影响；

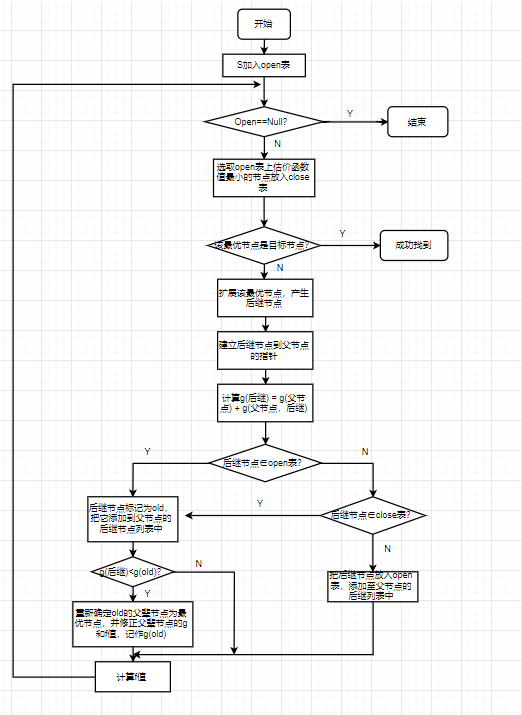
4、以八数码问题或路径规划等问题为例设计启发式搜索算法，改变启发函数，观察结果的变化，分析原因。

**四、 实验记录**

**搜索策略实验报告表一**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 姓名 |  | 年级 |  | 指导老师 |  | 日期 |  |
| 实验目的 | 熟悉和掌握各种启发式搜索策略的思想，掌握A\*算法的定义、估价函数和算法过程，理解求解流程和搜索顺序 | | | | | | |
| 搜索图 |  | | | | | | |
| 算法比较 | 广度优先 | | A\* | | Best First | | |
| Open表 | Open{S}  Open{1,2}  Open{2,3,4}  Open{3,4,5,6}  Open{7,4,5,6}  Open{7,8,5,6}  Open{7,8,9,G,6}  Open{7,8,9,G,10}  Open{8,9,G,10}  Open{9,G,10}  Open{G,10} | | Open{S}  Open{1,2}  Open{1,5,6}  Open{1,9,G,6} | | Open{S}  Open{1,2}  Open{1,5,6}  Open{1,5,10}  Open{1,5}  Open{1,9,G} | | |
| Close表 | S,1,2,3,4,5,6,7,8,9,10 | | S,2,5,G | | S,2,6,10,5,G | | |
| 估价函数 | F(x)=Depth | | F(x)=g(x)+h(x) | | F(x)=h(x) | | |
| 搜索节点次序记录 | S,1,2,3,4,5,6,7,8,9,10 | | S,2,5,G | | S,2,6,10,5,G | | |
| 观测结果 |  | |  | |  | | |
| 学生结论 | 广度优先搜索算法以接近起始节点的程度依次扩展节点，具有完备性，能找到最优解，但时间复杂度和空间复杂度较高，常常速度比较慢。 | | A\*算法综合考虑了从起始节点到当前节点的代价以及从当前节点到目标节点的启发信息。有了合适的估价函数，A\*算法能很快找到目标节点。A\*算法中，启发函数H(X)≤H\*(X)，具有完备性，能找到最优解。 | | 贪心算法利用了启发式信息，能够较快找到解，但是是不完备的，不一定得到最优解。 | | |

### 启发式搜索A\*算法框图



## 路径规划问题中的启发函数

在之前的作业中我们就已经写过A\*算法的程序代码。在八数码路径规划问题中，利用A\*算法去找出一条最短路，最要关注的就是估价函数，在本实验中，估价函数为路径代价g和启发函数h之和。进而我们需要关注启发函数。

在原启发函数的定义用该点到目标点的曼哈顿距离估计从该点到目标节点的代价。

源节点 目标节点

7 2 4 0 1 2

5 0 6 3 4 5

8 3 1 6 7 8

源程序如下：

def setH(self, endNode):

for x in range(0, 3):

for y in range(0, 3):

for m in range(0, 3):

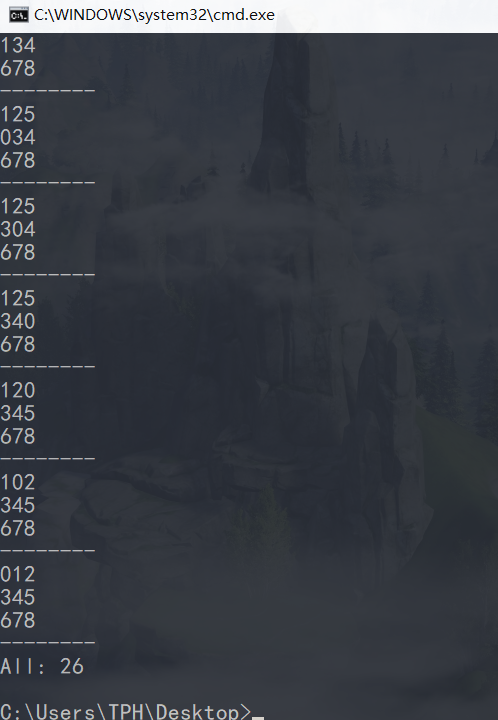
for n in range(0, 3):

if self.array2d[x][y] == endNode.array2d[m][n]:

self.h += abs(x-m)+abs(y-n)

上图中的多层循环意在取值该节点到目标节点的曼哈顿距离。

而在**曼哈顿距离**下的花销如下：



一共需要26步完成。并且程序执行速度也比较快。

**欧式距离**作为启发式

def setH(self, endNode):

for x in range(0, 3):

for y in range(0, 3):

for m in range(0, 3):

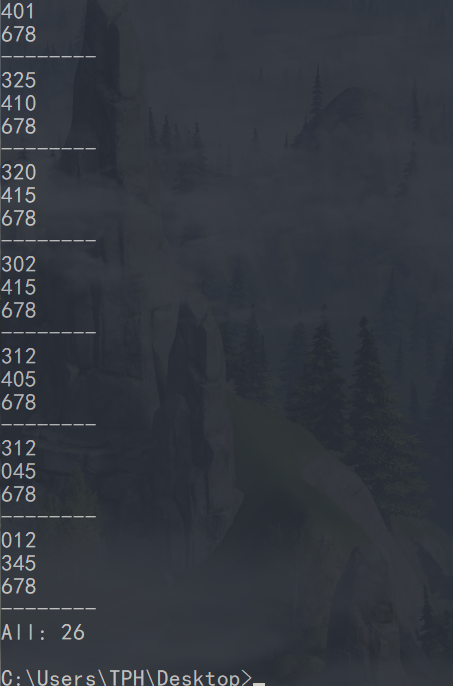
for n in range(0, 3):

if self.array2d[x][y] == endNode.array2d[m][n]:

#self.h += abs(x-m)+abs(y-n)

self.h += (abs(x-m)\*abs(x-m) + abs(y-n)\*abs(y-n)) \*\* 0.5

用欧式距离代替曼哈顿距离发现

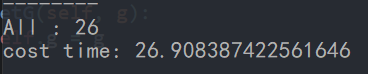


同样是26步，可以知道26步是该情况解的最优解。虽然两种启发式都能找到最优解，但是执行的速率是有差异的。如下搜索花费时间对比：

**曼哈顿：**



**欧氏距离：**



另外还尝试了**“错位棋子个数”**作为启发式函数

def setH(self,endNode):

for x in range(0,3):

for y in range(0,3):

if self.array2d[x][y] != endNode.array2d[x][y]:

self.h += 1

但是在此情况下，执行很久都没有执行完毕，起初怀疑程序有问题，但是更换了源节点、目的节点后发现3种启发式函数都较快找到了最优解。这说明，在特定的情况下应该选择特定的启发式函数，而不能一概而论，在老师给出的测试用例下，错位这种启发式搜索过程中产生的状态过于庞大，计算量太大，不易求解。