

A-star算法

——(N^2-1)数码的python实现

课程名称： 人工智能导论

小组编号： G03

小组成员： 刘书宇 31801323

小组成员： 童峻涛 31801341

专业班级： 软件工程1802

所在学院： 计算机与计算科学学院

报告日期： 2020 年 11 月 04 日

##### 实验目的

熟悉和掌握启发式搜索的定义、估价函数和算法过程，并利用A-star算法求解N数码难题，理解求解流程和搜索顺序。

以8数码问题和15数码问题为例实现A-star算法的求解程序（python为例），设计两种不同的估价函数（不在位数字的个数、曼哈顿距离）。

##### 问题简介

##### 问题背景

八数码问题也称作九宫问题，拼图问题。在3\*3的棋盘上，摆出八个棋子，每个棋子标有1~8中的某个数字，不同棋子上的数字不同，用0代替空格。棋盘上存在一个空格，与空格相邻的棋子可以移动到空格中。给出一个初始的棋子摆放状态，要求找出一种移动棋子步数最少的最优解，达到目的的棋子摆放状态。

十五数码问题来源于美国的科学魔术大师萨姆·洛伊德，洛伊德的发明其实只是将重排九宫（即8数码问题）中的3 阶方阵扩大到4 阶方阵罢了。由于这个细微的变化，十五数码问题的规模远远大于8数码问题，8数码问题的规模较小，总的状态数与15数码的状态数相差了8个数量级。

##### 解的存在性

在判断8数码和15数码是否存在解的判定中，采取逆序数之和奇偶性判断。结论可以简单表示为：

1. 将一个状态表示为一维的形式，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。
2. 若两个状态的逆序奇偶性相同，则可互相到达，否则不可相互到达。

##### 算法简介

##### 算法描述

A\*算法是一种求解最短路径最有效的直接搜索算法，也是目前最有影响的常用启发式算法。

定义 H\*(n) 为状态 n 到目的状态的最优路径的代价，则当 A 搜索算法的启发函数 H(n) 小于等于 H\*(n)，即满足：



A\*算法的启发函数为 F(n) = G(n) + H(n)，其中 F(n) 是从初始状态经由状态n到目标状态总代价的估计值，G(n) 是衡量某一状态在图中的深度（通俗的说就是当前已经走的步数），H(n)是从状态n到目标状态的最佳路径的估计代价。（在这里特别说明的是未带 \* 为评估代价但并不一定是最优代价）

则可以定义最优估价函数：



其中 h 是需要自己定义的，如果我们采用曼哈顿距离算法，具体公式为：



如果采用不在位数字的个数，则具体公式为：



##### 算法伪代码

1. open=[Stat]
2. closed=[]
3. **while** open不为空{
4. 从open中取出估价值f最小的节点n
5. **if** n == Target
6. **return** 从Stat到n的路径  //找到了！！！
7. **else**{
8. **for** n的每个子节点x{
9. **if** x in open{
10. 计算新的f(x)
11. 比较open表中的旧f(x)和新f(x)
12. **if** 新f(x) < 旧f(x){
13. 删掉open表里的旧x，加入新x
14. }
15. }
16. **else** **if** x in closed{
17. 计算新的f(x)
18. 比较closed表中的旧f(x)和新f(x)
19. **if** 新f(x) < 旧f(x){
20. remove x from closed
21. add x to open
22. }
23. }
24. **else** {
25. 计算f(x) add x to open
26. }
27. }
28. add n to closed
29. }
30. }

##### 样例解析

##### 以八数码（曼哈顿距离为估价函数）为例

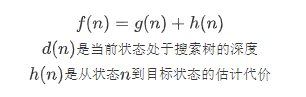
**初始状态：**

|  |  |  |
| --- | --- | --- |
| **2** | **8** | **3** |
| **1** | **6** | **4** |
| **7** |  | **5** |

**目标状态：**

|  |  |  |
| --- | --- | --- |
| **1** | **2** | **3** |
| **8** |  | **4** |
| **7** | **6** | **5** |

搜索树的估价函数为：



1. **初始化**

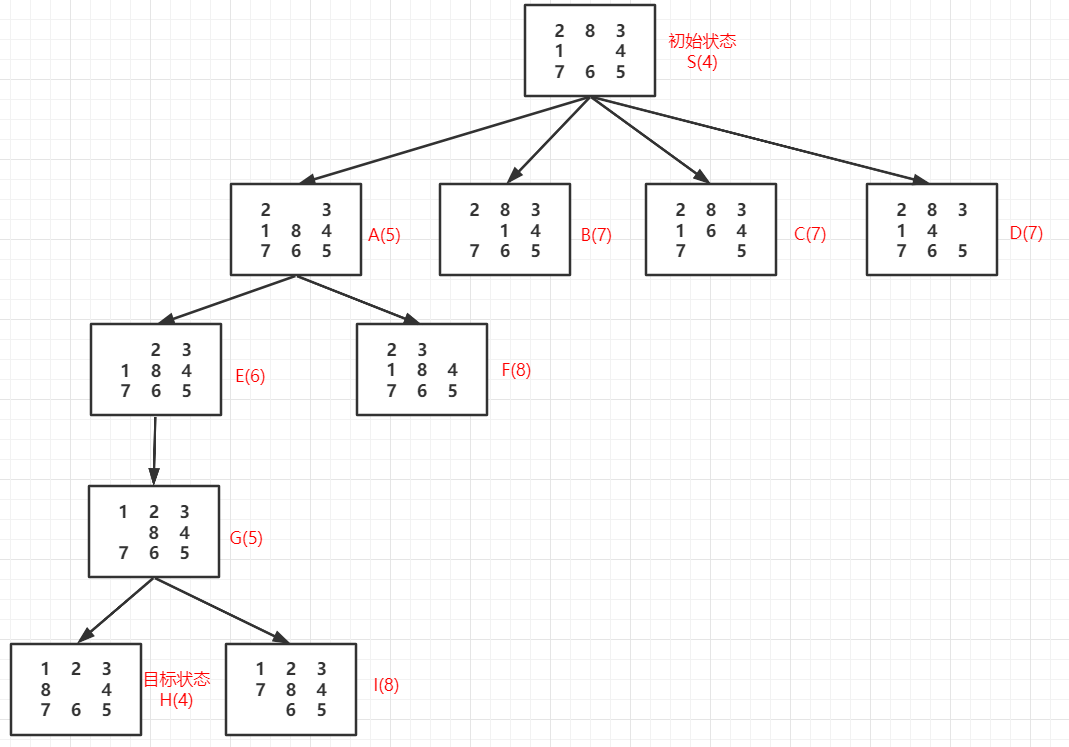
初始状态为 S(4)，其中 g(n) 等于当前状态位于搜索树的深度0，h(n) 等于当前状态到达目标状态的估计代价4，所以 0 + 4 = 4；

1. **第一次循环**
   1. 从Open表中取出第一个代价最小的状态 S(4)，如果该状态是目的状态，则搜索结束并返回Closed表；如果没有，则继续循环；
   2. 空白区域可以由上左下右四个方向的数字填补，通过上述的计算方法可以得到 A(5)、B(7)、C(7)、D(7) 四个状态；
   3. 将 A(5)、B(7)、C(7)、D(7) 四个状态归入Open表中按照每个状态的总代价升序排列，并将上一步状态 S(4) 归入Closed表中；
2. **第二次循环**
   1. 从Open表中取出第一个代价最小的状态 A(5)，如果该状态是目的状态，则搜索结束并返回Closed表；如果没有，则继续循环；
   2. 空白区域可以由左右四个方向的数字填补，因为从下数字补填的状态已经出现在了Open表或Closed表中，通过上述的计算方法可以得到 E(6)、F(7) 两个状态；
   3. 将 E(6)、F(8) 两个状态归入Open表中按照每个状态的总代价升序排列，并将上一步状态 A(5) 归入Closed表中；
3. **第三次循环**
   1. 从Open表中取出第一个代价最小的状态 E(6)，如果该状态是目的状态，则搜索结束并返回Closed表；如果没有，则继续循环；
   2. 空白区域只可以由下方的数字填补，因为从右边数字补填的状态已经出现在了Open表或Closed表中，通过上述的计算方法可以得到 G(5) 这个状态；
   3. 将 G(5) 状态归入Open表中按照每个状态的总代价升序排列，并将上一步状态 E(6) 归入Closed表中；
4. **第四次循环**
   1. 从Open表中取出第一个代价最小的状态 G(5)，如果该状态是目的状态，则搜索结束并返回Closed表；如果没有，则继续循环；
   2. 空白区域只可以由下方和右边的数字填补，因为从上边数字补填的状态已经出现在了Open表中或Closed表中，通过上述的计算方法可以得到 H(4)、I(8) 两个状态；
   3. 将 H(4)、I(8) 状态归入Open表中按照每个状态的总代价升序排列，并将上一步状态 G(5) 归入Closed表中；
5. **第五次循环**

从Open表中取出第一个代价最小的状态 H(4)，该状态就是目的状态，停止搜索并返回Closed表。

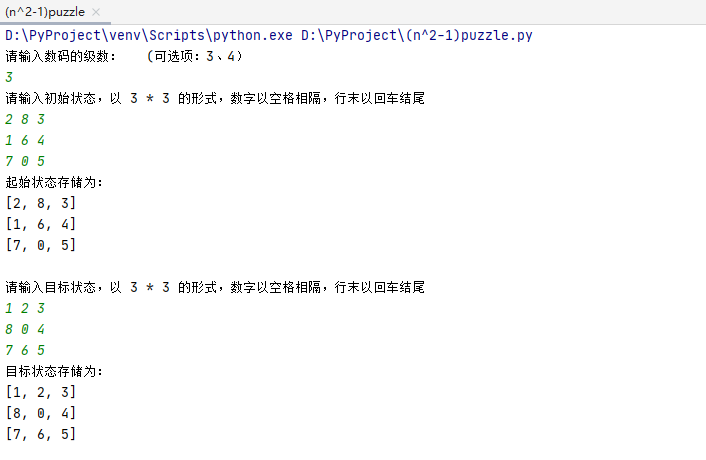
|  |  |
| --- | --- |
| **Open表** | **Closed表** |
| 初始化：( S(4) ) | ( ) |
| 一次循环后：( A(5), B(7), C(7), D(7) ) | ( S(4) ) |
| 二次循环后：( E(6), B(7), C(7), D(7), F(8) ) | ( S(4), A(5) ) |
| 三次循环后：( G(5), B(7), C(7), D(7), F(8) ) | ( S(4), A(5), E(6) ) |
| 四次循环后：( H(4), B(7), C(7), D(7), F(8), I(8) ) | ( S(4), A(5), E(6), G(5) ) |
| 五次循环后：H为目的状态，搜索成功 | ( S(4), A(5), E(6), G(5), H(4) ) |

**状态搜索树如下：**

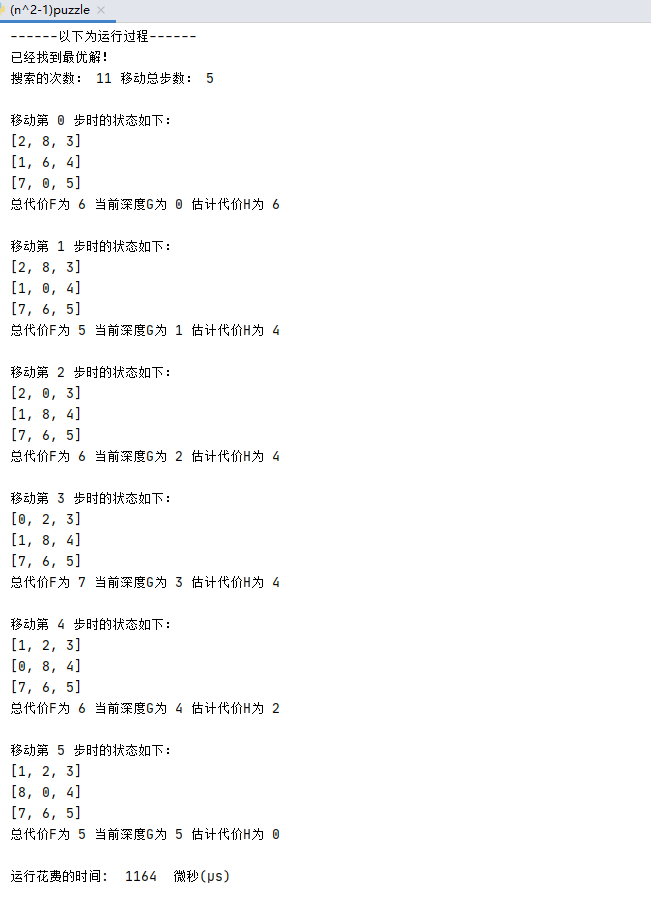


**运行结果如下：**

**输出部分：**



**运行部分：**



##### 实验总结

**表1 不同启发函数h（n）求解8数码问题的结果比较**

|  |  |  |
| --- | --- | --- |
|  | 启发函数h（n） | |
| 不在位数 | 曼哈顿距离 |
| 初始状态 | |  |  |  | | --- | --- | --- | | **2** | **8** | **3** | | **1** |  | **4** | | **7** | **6** | **5** | | |  |  |  | | --- | --- | --- | | **2** | **8** | **3** | | **1** |  | **4** | | **7** | **6** | **5** | |
| 目标状态 | |  |  |  | | --- | --- | --- | | **1** | **2** | **3** | | **8** |  | **4** | | **7** | **6** | **5** | | |  |  |  | | --- | --- | --- | | **1** | **2** | **3** | | **8** |  | **4** | | **7** | **6** | **5** | |
| 生成节点数 | 11 | 9 |
| 运行时间（微秒） | 1236us | 954us |

**表2 不同启发函数h（n）求解15数码问题的结果比较**

|  |  |  |
| --- | --- | --- |
|  | 启发函数h（n） | |
| 不在位数 | 曼哈顿距离 |
| 初始状态 | |  |  |  |  | | --- | --- | --- | --- | | 5 | 1 | 2 | 4 | | 9 | 6 | 3 | 8 | | 13 | 15 | 10 | 11 | | 14 |  | 7 | 12 | | |  |  |  |  | | --- | --- | --- | --- | | 5 | 1 | 2 | 4 | | 9 | 6 | 3 | 8 | | 13 | 15 | 10 | 11 | | 14 |  | 7 | 12 | |
| 目标状态 | |  |  |  |  | | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | | 5 | 6 | 7 | 8 | | 9 | 10 | 11 | 12 | | 13 | 14 | 15 |  | | |  |  |  |  | | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | | 5 | 6 | 7 | 8 | | 9 | 10 | 11 | 12 | | 13 | 14 | 15 |  | |
| 生成节点数 | 132 | 136 |
| 运行时间（微秒） | 15658us | 16011us |

通过设置相同的初始状态和目标函数，针对不同的估价函数，求解问题的解，比较这些参数对搜索算法性能的影响，可以得出曼哈顿距离优于不在位数的估价函数，可见估价函数的选取对于算法的运行存在很大的影响，这种影响在小型数据测试中不宜显现，而在大型数据的测试下会更加明显。此外，由于复杂度的原因十五数码的运算时间高于八数码。

由此我们还得出A\*启发式算法的特点，概括如下：

1. 完备性：肯定能找到最优解（除非不存在解）
2. 最优性：找到的解花费最小
3. 速度快：扩展更少的节点（取决于估价函数的选择）

##### 实验代码

1. **import** copy
2. **import** time
3. #初始状态
4. **print**("请输入数码的级数：   (可选项：3、4）")
5. n = int(input())
6. **print**("请输入初始状态，以",n,"\*",n,"的形式，数字以空格相隔，行末以回车结尾")
7. stat = [[0]\*n]\*n
8. **for** i **in** range(n):
9. stat[i] = input().split(" ")
10. stat[i] = [int(j) **for** j **in** stat[i]]
11. **print**("起始状态存储为：")
12. **for** i **in** range(n):
13. **print**(stat[i])
14. **print**()
15. **print**("请输入目标状态，以",n,"\*",n,"的形式，数字以空格相隔，行末以回车结尾")
16. target = [[0]\*n]\*n
17. **for** i **in** range(n):
18. target[i] = input().split(" ")
19. target[i] = [int(j) **for** j **in** target[i]]
20. **print**("目标状态存储为：")
21. **for** i **in** range(n):
22. **print**(target[i])
23. **print**()
24. **print**("------以下为运行过程------")
25. #棋盘的类，实现移动和扩展状态
26. **class** puzzle:
27. **def** \_\_init\_\_(self,stat,target):
28. self.pre=None
29. #目标状态
30. self.target=target
31. #stat是一个二维列表
32. self.stat=stat
33. self.find0()
34. self.update()
35. #更新启发函数的相关信息
36. **def** update(self):
37. self.fH()
38. self.fG()
39. self.fF()
41. #G是深度，也就是走的步数
42. **def** fG(self):
43. **if**(self.pre!=None):
44. self.G=self.pre.G+1
45. **else**:
46. self.G=0
48. #H是和目标状态距离之和 曼哈顿距离 或者为不在位数
49. **def** fH(self):
50. self.H=0
51. # 曼哈顿距离之和
52. **for** i **in** range(n):
53. **for** j **in** range(n):
54. targetX=self.target[i][j]
55. nowP=self.findx(targetX)
57. self.H+=abs(nowP[0]-i)+abs(nowP[1]-j)
59. # 不在位数
60. # for i in range(n):
61. #     for j in range(n):
62. #         targetX=self.target[i][j]
63. #         nowP=self.findx(targetX)
64. #         if(abs(nowP[0]-i)+abs(nowP[1]-j)>0):
65. #             self.H  = self.H+1

68. #F是启发函数，F=G+H
69. **def** fF(self):
70. self.F=self.G+self.H
72. #以四行四列的形式输出当前状态
73. **def** see(self):
74. **for** i **in** range(n):
75. **print**(self.stat[i])
76. **print**("总代价F为",self.F,"当前深度G为",self.G,"估计代价H为",self.H)
77. **print**()
78. #查看找到的解是如何从头移动的
79. **def** seeAns(self):
80. ans=[]
81. ans.append(self)
82. p=self.pre
83. **while**(p):
84. ans.append(p)
85. p=p.pre
86. ans.reverse()
87. time = 0;
88. **for** i **in** ans:
89. **print**("移动第",time,"步时的状态如下：")
90. time = time +1
91. i.see()
93. #找到数字x的位置
94. **def** findx(self,x):
95. **for** i **in** range(n):
96. **if**(x **in** self.stat[i]):
97. j=self.stat[i].index(x)
98. **return** [i,j]
100. #找到0，也就是空白格的位置
101. **def** find0(self):
102. self.zero=self.findx(0)
104. #扩展当前状态，也就是上下左右移动。返回的是一个状态列表，也就是包含stat的列表
105. **def** expand(self):
106. i=self.zero[0]  #x坐标
107. j=self.zero[1]  #y坐标
108. gridList=[]
109. #空白格纵坐标显示中右，默认向左
110. **if**(n==3):
111. **if** (j == 2 **or** j == 1 ):
112. gridList.append(self.left())
113. **if** (i == 2 **or** i == 1 ):
114. gridList.append(self.up())
115. **if** (i == 0 **or** i == 1 ):
116. gridList.append(self.down())
117. **if** (j == 0 **or** j == 1 ):
118. gridList.append(self.right())
119. **return** gridList
120. **if**(n==4):
121. **if** (j == 2 **or** j == 1 **or** j == 3):
122. gridList.append(self.left())
123. **if** (i == 2 **or** i == 1 **or** i == 3):
124. gridList.append(self.up())
125. **if** (i == 0 **or** i == 1 **or** i == 2):
126. gridList.append(self.down())
127. **if** (j == 0 **or** j == 1 **or** j == 2):
128. gridList.append(self.right())
129. **return** gridList


133. #deepcopy多维列表的复制，防止指针赋值将原列表改变
134. #move只能移动行或列，即row和col必有一个为0
135. #向某个方向移动
136. **def** move(self,row,col):
137. newStat=copy.deepcopy(self.stat)
138. tmp=self.stat[self.zero[0]+row][self.zero[1]+col]
139. newStat[self.zero[0]][self.zero[1]]=tmp
140. newStat[self.zero[0]+row][self.zero[1]+col]=0
141. **return** newStat
143. **def** up(self):
144. **return** self.move(-1,0)
146. **def** down(self):
147. **return** self.move(1,0)
149. **def** left(self):
150. **return** self.move(0,-1)
152. **def** right(self):
153. **return** self.move(0,1)
155. #判断状态g是否在状态集合中，g是对象，gList是对象列表
156. #返回的结果是一个列表，第一个值是真假，如果是真则第二个值是g在gList中的位置索引
157. **def** isin(g,gList):
158. gstat=g.stat
159. statList=[]
160. **for** i **in** gList:
161. statList.append(i.stat)
162. **if**(gstat **in** statList):
163. res=[True,statList.index(gstat)]
164. **else**:
165. res=[False,0]
166. **return** res
168. #计算逆序数之和
169. **def** N(nums):
170. N=0
171. nums = sum(nums, [])
172. **for** i **in** range(len(nums)):
173. **if**(nums[i]!=0):
174. **for** j **in** range(i):
175. **if**(nums[j]>nums[i]):
176. N+=1
177. **return** N
179. #根据逆序数之和判断所给八数码是否可解
180. **def** judge(src,target):
181. N1=N(src)
182. N2=N(target)
183. **if**(N1%2==N2%2):
184. **return** True
185. **else**:
186. **return** False
188. #Astar算法的函数
189. **def** Astar(startStat):
190. #open和closed存的是grid对象
191. open=[]
192. closed=[]
193. #初始化状态
194. g=puzzle(startStat,target)
195. #检查是否有解
196. **if**(judge(startStat,g.target)!=True):
197. **print**("所给样例无解，请检查输入")
198. exit(1)
200. open.append(g)
201. #time变量用于记录遍历次数
202. time=0
203. #当open表非空时进行遍历
204. **while**(open):
205. #根据启发函数值对open按照F进行排序，默认升序
206. open.sort(key=**lambda** G:G.F)
207. #找出启发函数值最小的进行扩展
208. minFStat=open[0]
209. #检查是否找到解，如果找到则从头输出移动步骤
210. **if**(minFStat.H==0):
211. **print**("已经找到最优解！")
212. **print**("搜索的次数：",time,"移动总步数：",minFStat.G)
213. **print**()
214. minFStat.seeAns()
215. **break**
217. #走到这里证明还没有找到解，对启发函数值最小的进行扩展
218. open.pop(0)
219. closed.append(minFStat)
220. expandStats=minFStat.expand()
221. #遍历扩展出来的状态
222. **for** stat **in** expandStats:
223. #将扩展出来的状态（二维列表）实例化为grid对象
224. tmpG=puzzle(stat,target)
225. #指针指向父节点
226. tmpG.pre=minFStat
227. #初始化时没有pre，所以G初始化时都是0
228. #在设置pre之后应该更新G和F
229. tmpG.update()
230. #查看扩展出的状态是否已经存在与open或closed中
231. findstat=isin(tmpG,open)
232. findstat2=isin(tmpG,closed)
233. #在closed中,判断是否更新
234. **if**(findstat2[0]==True **and** tmpG.F<closed[findstat2[1]].F):
235. closed[findstat2[1]]=tmpG
236. open.append(tmpG)
237. time+=1
238. #在open中，判断是否更新
239. **if**(findstat[0]==True **and** tmpG.F<open[findstat[1]].F):
240. open[findstat[1]]=tmpG
241. time+=1
242. #tmpG状态不在open中，也不在closed中
243. **if**(findstat[0]==False **and** findstat2[0]==False):
244. open.append(tmpG)
245. time+=1
247. start = time.perf\_counter()
248. #long running
249. Astar(stat)
251. elapsed = (time.perf\_counter() - start)
252. **print**("运行花费的时间: ",(int)(elapsed\*1000\*1000)," 微秒(μs)")

##### 附录

八数码测试数据：

3

2 8 3

1 0 4

7 6 5

1 2 3

8 0 4

7 6 5

十五数码测试数据：

4

5 1 2 4

9 6 3 8

13 15 10 11

14 0 7 12

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 0