

搜索策略实验报告

|  |  |
| --- | --- |
| 学生姓名 | maybeLocalhost |
| 学 号 |  |
| 专业班级 |  |
| 指导教师 |  |
| 学 院 | 计算机学院 |
| 完成时间 | 2020.11 |

目录

[一、实验内容 1](#_Toc60250763)

[二、实验目的 1](#_Toc60250764)

[三、 实验内容 1](#_Toc60250765)

[四、实验环境 1](#_Toc60250766)

[五、实验步骤 2](#_Toc60250767)

[六、搜索策略实验报告表 2](#_Toc60250768)

[七、实验结论 5](#_Toc60250769)

[附：A\*搜索算法代码 7](#_Toc60250770)

# 一、实验内容

1. 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。
2. 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、实验环境

以下两种实验环境可供选择：

1. 搜索策略可视化实验环境，如图1所示。

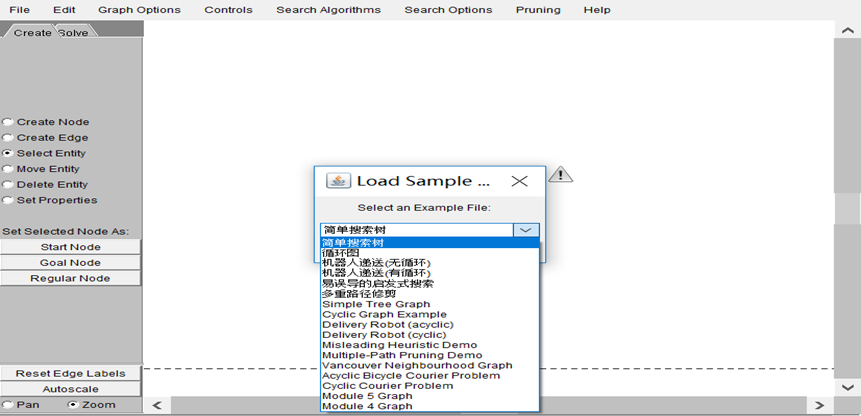


图 1 搜索策略可视化实验环境

1. C＋＋语言编程环境，语言环境可以自选。

# 五、实验步骤

1. 开始演示。进入搜索策略演示程序，可从多种不同搜索算法选择装载相关源文件；
2. 选择不同的搜索算法，点击“autosearch” 观察搜索过程；
3. 设置不同属性，观察搜索过程的变化；
4. 观察运行过程和搜索顺序，理解启发式搜索的原理；
5. 算法流程的任一时刻的相关状态,以算法流程高亮、open表、close表、节点静态图、当前扩展节点移动图等5种形式在按钮上方同步显示,便于深入学习理解搜索算法。
6. 根据程序运行过程画出搜索算法框图；
7. 若要自己设计改进算法并运行，可参考帮助文件。

# 六、搜索策略实验报告表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 王启乐 | 年级 | 信安1802班 | 指导老师 | 钟萍 | 日期 | 2020.11.18 |
| 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 | 广度优先搜索 | | 最佳优先搜索 | | A\*算法 | | |
| Open表 | {初始节点 S}  {节点 1，节点 2}  {节点 2，节点 3，节点 4}  {节点 3，节点 4，节点 5，节点6}  {节点 4，节点 5，节点 6，节点7}  {节点 5，节点 6，节点 7，节点8}  {节点 6，节点 7，节点 8，节点 9  目标节点 G}  {节点 7，节点 8，节点 9，目标节点 G，节点 10}  {节点 8，节点 9，目标节点 G，节点 10}  {节点 9，目标节点 G，节点 10}  {目标节点 G，节点 10}  {节点 10} | | {初始节点 S}  {节点 2，节点 1}  {节点 6，节点 5，节点 1}  {节点 10，节点 5，节点 1}  {节点 5，节点 1}  {目标节点 G，节点 9，节点 1}  {节点 9，节点 1} | | {初始节点 S}  {节点 2，节点 1}  {节点 1，节点 5，节点 6}  {节点 5，节点 6，节点 4，节点 3}  {节点 6，节点 4，目标节点 G，节点9，节点 3}  {节点 4，目标节点 G，节点 9，节点3，节点 10}  {目标节点 G，节点 9，节点 3，节点10，节点 8}  {节点 9，节点 3，节点 10，节点 8} | | |
| Close表 | {}  {初始节点 S}  {初始节点 S，节点 1}  {初始节点 S，节点 1，节点 2}  {初始节点 S，节点 1，节点 2，节点 3}  {初始节点 S，节点 1，节点 2，节点 3，节点 4}  {初始节点 S，节点 1，节点 2，节点 3，节点 4，节点 5}  {初始节点 S，节点 1，节点 2，节点 3，节点 4，节点 5，节点 6}  {初始节点 S，节点 1，节点 2，节点 3，节点 4，节点 5，节点 6， 节点 7}  {初始节点 S，节点 1，节点 2，节点 3，节点 4，节点 5，节点 6， 节点 7，节点 8}  {初始节点 S，节点 1，节点 2，节点 3，节点 4，节点 5，节点 6， 节点 7，节点 8，节点 9}  {初始节点 S，节点 1，节点 2，节点 3，节点 4，节点 5，节点 6， 节点 7，节点 8，节点 9，目标节点 G} | | {}  {初始节点 S}  {初始节点 S，节点 2}  {初始节点 S，节点 2，节点 6}  {初始节点 S，节点 2，节点 6， 节点 10}  {初始节点 S，节点 2，节点 6， 节点 10，节点 5}  {初始节点 S，节点 2，节点 6， 节点 10，节点 5，目标节点 G} | | {}  {初始节点 S}  {初始节点 S，节点 2}  {初始节点 S，节点 2，节点 1}  {初始节点 S，节点 2，节点 1，节点5}  {初始节点 S，节点 2，节点 1，节点5，节点 6}  {初始节点 S，节点 2，节点 1，节点5，节点 6，节点 4}  {初始节点 S，节点 2，节点 1，节点5，节点 6，节点 4，目标节点 G} | | |
| 估价函数 | f(n)为节点 n 的深度 | | f(n) = h(n)  h(n)为估计的从n 到目标的最优距离 | | f(n) = g(n) + h(n)  g(n)为搜索树中S 到n 这段路径的代价，h(n) 为估计的从 n 到目标的最优距离 | | |
| 搜索节点次序记录 | {初始节点 S，节点 1，节点 2，节点 3，节点 4，节点 5，节点 6， 节点 7，节点 8，节点 9，目标节点 G} | | {初始节点 S，节点 2，节点 6， 节点 10，节点 5，目标节点 G} | | {初始节点 S，节点 2，节点 1，节点5，节点 6，节点 4，目标节点 G} | | |
| 观测结果 |  | |  | |  | | |
| 学生结论 | 广度优先搜索算法搜索成功，但扩展的节点数较多搜索速度较慢，其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。 | | 最佳优先搜索算法搜索成功，扩展的节点数少，搜索速度较快，它是一种不追求最优解，只希望求得较为满意的解。因为他省去了为了找到最优解要穷尽所有可能而必须耗费的大量时间。 | | A\*算法搜索成功，扩展的节点数较少，搜索速度较快，它是一种静态路网中求解最短路径最有效的直接搜索方法，也是解决许多搜索问题的有效算法。算法中的距离估算值与实际值越接近，最终搜索速度越快。 | | |

# 七、实验结论

1. **启发式搜索算法A\*流程图和算法框图；**

A\*算法是一种启发式搜索，评估方程是：f(n) = d(n)+h(n)，当OPEN表不为空时，每次循环从OPEN表中取出评估值最小的状态节点进行扩展，判断该节点是否为目标节点，如果是则输出移动的路径；如果不是则将子节点放入OPEN表。并且为了避免状态的重复搜索，用hash函数来判断状态是否存在，如果已存在则不会扩展该节点。

A\*算法流程图如下：

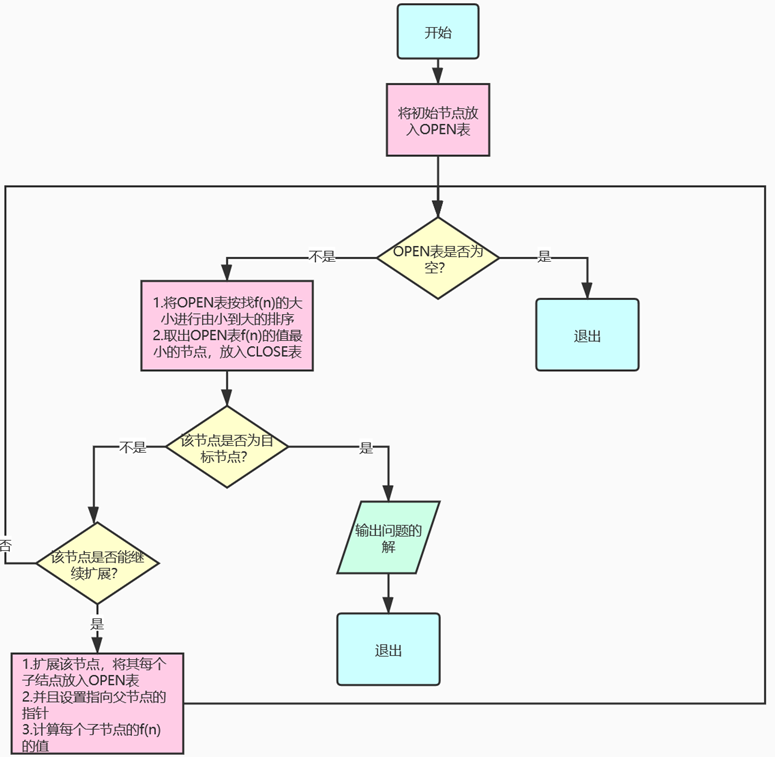


图 2 A\*算法流程图

1. **试分析估价函数的值对搜索算法速度的影响；**

估价函数能够提供一个评定候选扩展结点的方法，以确定哪个节点最有可能在通向目标的最佳路径上。估价函数构造得越准确，就越快能找到最应该被扩展的节点，搜索算法的速度就会越快。

以路径规划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。

修改前的启发函数：

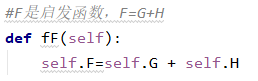


图 3 修改前的启发函数

运行结果：

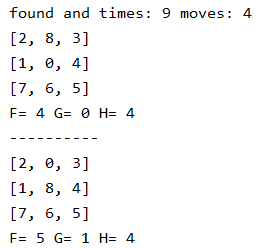


图 4 修改前的启发函数的运行结果

修改后的启发函数：

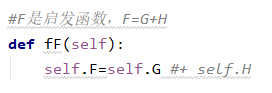


图 5 修改后的启发函数

运行结果：

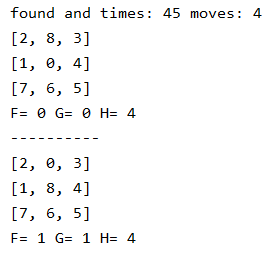


图 6 修改后的启发函数的运行结果

可以观察到，修改后的启发函数导致搜索算法速度明显变慢。

1. **根据A\*算法分析启发式搜索的特点。**

启发式搜索算法，就是在状态空间中的搜索对每一个搜索的位置进行评估，得到最好的位置，再从这个位置进行搜索直到目标。A\*算法把从初始节点到n 节点的实际代价g(n)和从n 到目标节点最佳路径的估计代价h(n)结合起来对节点n 进行评价，从而确定下一个被扩展的节点。

# 附：A\*搜索算法代码

1. **import** copy
2. #棋盘的类，实现移动和扩展状态
3. **class** grid:
4. **def** \_\_init\_\_(self,stat):
5. self.pre=None
6. #stat是一个二维列表
7. self.stat=stat
8. self.find0()
9. self.update()
11. #更新启发函数的相关信息
12. **def** update(self):
13. self.fH()
14. self.fG()
15. self.fF()
17. #G是深度，也就是走的步数
18. **def** fG(self):
19. **if**(self.pre!=None):
20. self.G=self.pre.G+1
21. **else**:
22. self.G=0
24. #H是和目标状态距离之和
25. **def** fH(self):
26. target=[[1,2,3],[8,0,4],[7,6,5]]
27. self.H=0
28. **for** i **in** range(3):
29. **for** j **in** range(3):
30. targetX=target[i][j]
31. nowP=self.findx(targetX)
32. #曼哈顿距离之和
33. self.H+=abs(nowP[0]-i)+abs(nowP[1]-j)
35. #F是启发函数，F=G+H
36. **def** fF(self):
37. self.F=self.G #+ self.H
39. #以三行三列的形式输出当前状态
40. **def** see(self):
41. **for** i **in** range(3):
42. **print**(self.stat[i])
43. **print**("F=",self.F,"G=",self.G,"H=",self.H)
44. **print**("-"\*10)
46. #查看找到的解是如何从头移动的
47. **def** seeAns(self):
48. ans=[]
49. ans.append(self)
50. p=self.pre
51. **while**(p):
52. ans.append(p)
53. p=p.pre
54. ans.reverse()
55. **for** i **in** ans:
56. i.see()
58. #找到数字x的位置
59. **def** findx(self,x):
60. **for** i **in** range(3):
61. **if**(x **in** self.stat[i]):
62. j=self.stat[i].index(x)
63. **return** [i,j]
65. #找到0，也就是空白格的位置
66. **def** find0(self):
67. self.zero=self.findx(0)
69. #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
70. **def** expand(self):
71. i=self.zero[0]
72. j=self.zero[1]
73. gridList=[]
74. **if**(j==2 **or** j==1):
75. gridList.append(self.left())
76. **if**(i==2 **or** i==1):
77. gridList.append(self.up())
78. **if**(i==0 **or** i==1):
79. gridList.append(self.down())
80. **if**(j==0 **or** j==1):
81. gridList.append(self.right())
82. **return** gridList

85. #deepcopy多维列表的复制，防止指针赋值将原列表改变
86. #move只能移动行或列，即row和col必有一个为0
87. #向某个方向移动
88. **def** move(self,row,col):
89. newStat=copy.deepcopy(self.stat)
90. tmp=self.stat[self.zero[0]+row][self.zero[1]+col]
91. newStat[self.zero[0]][self.zero[1]]=tmp
92. newStat[self.zero[0]+row][self.zero[1]+col]=0
93. **return** newStat
95. **def** up(self):
96. **return** self.move(-1,0)
98. **def** down(self):
99. **return** self.move(1,0)
101. **def** left(self):
102. **return** self.move(0,-1)
104. **def** right(self):
105. **return** self.move(0,1)
107. #判断状态g是否在状态集合中，g是对象，gList是对象列表
108. #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
109. **def** isin(g,gList):
110. gstat=g.stat
111. statList=[]
112. **for** i **in** gList:
113. statList.append(i.stat)
114. **if**(gstat **in** statList):
115. res=[True,statList.index(gstat)]
116. **else**:
117. res=[False,0]
118. **return** res
120. #Astar算法的函数
121. **def** Astar(startStat):
122. #open和closed存的是grid对象
123. open=[]
124. closed=[]
125. #初始化状态
126. g=grid(startStat)
127. open.append(g)
128. #time变量用于记录遍历次数
129. time=0
130. #当open表非空时进行遍历
131. **while**(open):
132. #根据启发函数值对open进行排序，默认升序
133. open.sort(key=**lambda** G:G.F)
134. #找出启发函数值最小的进行扩展
135. minFStat=open[0]
136. #检查是否找到解，如果找到则从头输出移动步骤
137. **if**(minFStat.H==0):
138. **print**("found and times:",time,"moves:",minFStat.G)
139. minFStat.seeAns()
140. **break**
142. #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
143. open.pop(0)
144. closed.append(minFStat)
145. expandStats=minFStat.expand()
146. #遍历扩展出来的状态
147. **for** stat **in** expandStats:
148. #将扩展出来的状态（二维列表）实例化为grid对象
149. tmpG=grid(stat)
150. #指针指向父节点
151. tmpG.pre=minFStat
152. #初始化时没有pre，所以G初始化时都是0
153. #在设置pre之后应该更新G和F
154. tmpG.update()
155. #查看扩展出的状态是否已经存在与open或closed中
156. findstat=isin(tmpG,open)
157. findstat2=isin(tmpG,closed)
158. #在closed中,判断是否更新
159. **if**(findstat2[0]==True **and** tmpG.F<closed[findstat2[1]].F):
160. closed[findstat2[1]]=tmpG
161. open.append(tmpG)
162. time+=1
163. #在open中，判断是否更新
164. **if**(findstat[0]==True **and** tmpG.F<open[findstat[1]].F):
165. open[findstat[1]]=tmpG
166. time+=1
167. #tmpG状态不在open中，也不在closed中
168. **if**(findstat[0]==False **and** findstat2[0]==False):
169. open.append(tmpG)
170. time+=1
172. stat=[[2, 8, 3], [1, 0 ,4], [7, 6, 5]]
173. Astar(stat)