基于迷宫问题的算法探究

小组成员及分工：

22009201386 马 龙 算法编程及优化

22009200719 白玉安 论文撰写

**摘要：**迷宫问题是一种经典的数学问题，也被称为迷宫寻路问题，通常是指在一个二维迷宫中寻找一条从入口到出口的路径。迷宫问题有多种解决方法，包括深度优先搜索（DFS）、广度优先搜索（BFS）和 A\*算法等。本文通过深度优先搜索 (DFS)和 A\*算法来研究解决迷宫问题的方法。同时，给出算法核心代码框架，并进行算法优化，绘出可视化思维模型。

**关键词：**算法 迷宫问题 可视化

**一、理论背景及知识**

**1.DFS算法基本原理**

深度优先搜索算法（DFS）主要的搜索方法就是以“深度”为核心。深度优先搜索的基本思想是:从初始节点S。开始，在其子节点中进行考查，若不是目标节点，则再在该子节点的子节点中选择一个节点进行考查，一直如此向下搜索。当到达某个子节点，若孩子节点既不是目标节点又不能继续扩展，则选择其兄弟节点进行考查。搜索过程如下。

步骤一：把初始节点S，放入OPEN表。

步骤二：如果OPEN表为空，则问题无解，失败退出，否则继续。

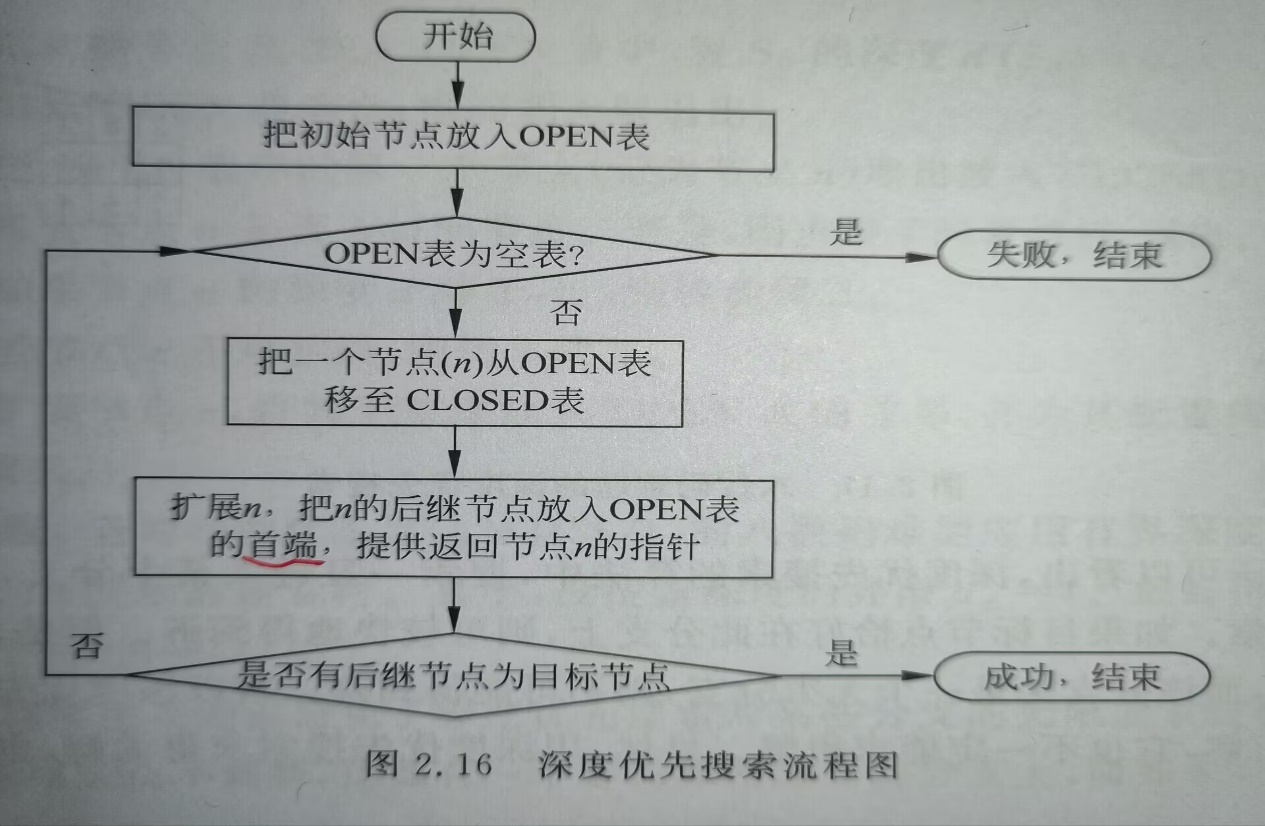
步骤三：把第一个节点(记为节点n)从OPEN表移出，并把它放人已扩展节点表(CL0SED表)中。

步骤四：考查节点n是否为目标节点。若是，则得到问题的解，退出。

步骤五：若节点n不可扩展，则转步骤二。

步骤六：扩展节点n，将其子节点放入人OPEN表的首部，并为其配置指向父节点的指针，然后转向步骤二。

深度优先搜索算法流程如下图所示。



**2.A\*算法基本原理**

A\*算法是一种启发式搜索算法，用于在二维或多维空间中找到从起始点到目标点的最短路径。该算法结合了深度优先搜索（DFS）和广度优先搜索（BFS）的优点，通过使用启发函数来评估每个节点的优先级，从而避免了盲目搜索和重复搜索。

在A\*算法中，往往通过一个函数来计算每个节点的权值，而这个函数，则决定算法的搜索效率。这里定义函数z=x+y，其中z表示从开始到结束点之间的距离预估；x表示从起始节点到节点v的实际距离；y表示从节点v到目标节点的最短距离预估。A\*算法需要两个状态表：开放列表（Open表）和闭合列表（Closed表）。

A\*算法的具体搜索步骤为：

1. 选取s作为起始节点，将其标记为待考察并放入开放列表中；

2. 以 s 为当前节点，搜索其所有相邻节点 v 标记为待考察放入开放列表中（无法到 达的节点不考虑在内），将 s 标记为已考察，放入闭合列表中；

3. 设置v的父节点为s，并计算z、x、y值；

4. 如果开放列表为空，则寻路失败；反之开放列表不为空，查询其中z值中值最小的 点a，然后其移入闭合列表；

5. 如果a为目标节点，则寻路成功；如果a不为目标节点，则转到第6步；

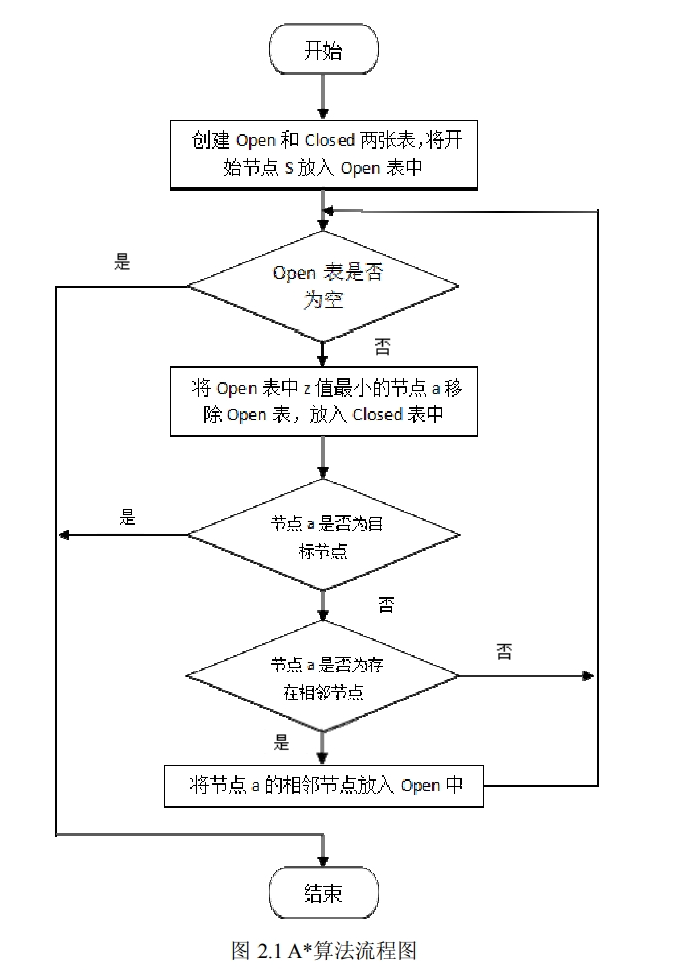
6. 以a为当前节点，搜索其与a点领近的点b（无法到达或者在闭合列表中不考虑在 内），假如搜索到的b点没有在开放列表中，将将它们放入；

7. 设置b的父节点为a，并计算b的z、x、y值。如果某个b在开放列表中，则转到第8 步；

8. 检查经过a到达b的路径，x值是否减少，如果x减少，则把b的父节点改为v，然后 重新计算z、x、y值；如果x值没有减少，则无需操作；

9. 转到第4步。

A\*算法流程如下图所示。

****

**二、深度优先算法解决迷宫问题**

**1．问题建模**

假设我们有一个二维迷宫，其中每个方块可以是墙壁或空地。迷宫的起始位置是(0,0)，目标位置是(n,m)，其中n和m是迷宫的大小。

**2. 问题解决**

利用深度优先算法解决迷宫问题的一般步骤如下：

1.建立迷宫：建立一个二维数组来表示迷宫。0表示道路，1表示墙壁。

2.初始化方向和路径：设定一个方向数组，用于记录从当前位置出发，四个基本方向（上下左右）是否可行。同时，初始化一个路径数组，用于存储当前路径。

3.定义移动逻辑：编写一个函数，用于实现从当前位置出发，按照深度优先算法探索相邻单元格。这个函数需要判断相邻单元格是否为墙壁、是否已经访问过、以及当前方向是否可行。如果可行，则更新路径数组，并将当前方向标记为已访问。

4.查找出口：从起点开始，按照深度优先算法探索迷宫。在探索过程中，当遇到终点时，说明找到了一条路径。此时，可以返回路径数组，逆序还原出从起点到终点的最短路径。

5.优化路径：在找到路径后，可以对路径进行优化。

以下是一些简单的示例代码：

文件名：main.py

源代码：

1. **from** maze **import** main
3. **import** numpy as np
5. x=np.array([[0,0,0,1],[1,1,0,0],[1,1,1,0]])
7. main(x)

文件名：canWide.py

源代码：

1. **import** numpy as np
3. **def** canWiden(cur,arr):
4. #cur是一个列表,包含两个元素,第一个元素是当前位置行索引,第二个元素是列索引
6. #需要返回的可拓展结点索引的列表
7. res=[]
9. #arr的行数,列数
10. row=np.shape(arr)[0]
11. col=np.shape(arr)[1]
13. #起点也标记为-1
14. arr[0,0]=-1
16. #按照向右,向下,向左,向上的顺序拓展结点,并将可拓展结点标记为-1
17. #此外还需要将可拓展结点加入列表res
18. **if** cur[1]+1<col **and** arr[cur[0],cur[1]+1]==0:
19. res.append([cur[0],cur[1]+1])
20. arr[cur[0],cur[1]+1]=-1
22. **if** cur[0]+1<row **and** arr[cur[0]+1,cur[1]]==0:
23. res.append([cur[0]+1,cur[1]])
24. arr[cur[0]+1,cur[1]]=-1
26. **if** cur[1]-1>=0 **and** arr[cur[0],cur[1]-1]==0:
27. res.append([cur[0],cur[1]-1])
28. arr[cur[0],cur[1]-1]=-1
30. **if** cur[0]-1>=0 **and** arr[cur[0]-1,cur[1]]==0:
31. res.append([cur[0]-1,cur[1]])
32. arr[cur[0]-1,cur[1]]=-1
34. **return** res

文件名：maze.py

源代码：

1. #使用numpy中的ndarray对象来表示迷宫
2. **import** numpy as np
4. #其中[0,0]元素为起点,对角元素为终点
6. #导入自己编写的栈类型
7. **from** stack **import** stack
9. #拓展结点函数导入
10. #canWiden函数接受两个参数,第一个参数是当前位置的索引,第二个参数是ndarray对象,即迷宫
11. **from** canWiden **import** canWiden
13. **from** printPath **import** printPath
15. #主函数
16. **def** main(x):
17. #参数x就是迷宫
19. #OPEN表,CLOSED表
20. OPEN=stack()
21. CLOSED=stack()
23. #当前结点索引
24. cur=[0,0]
26. #目标位置索引
27. tar=[np.shape(x)[0]-1,np.shape(x)[1]-1]

30. #开始深度优先搜索
31. #起点索引放入OPEN表
32. OPEN.push(cur)
34. **while** tar **not** **in** OPEN.all():#all()函数以列表形式返回OPEN表中所有元素
35. #OPEN为空时不再遍历
36. **if** OPEN.isempty():
37. **break**
39. #OPEN表第一个元素放入CLOSED表中
40. cur=OPEN.pop()
41. CLOSED.push(cur)
43. #将可拓展结点放入OPEN表
44. wideNodes=canWiden(cur,x)
45. **for** i **in** wideNodes:
46. OPEN.push(i)
48. printPath(CLOSED.all(),tar)

文件名：printPath.py

源代码：

1. **def** printPath(CLOSED:list,tar:list):
2. #CLOSED是CLOSED表中所有元素
4. #tar是目标位置索引
6. #如果tar结点的左边结点或者上边结点在CLOSED表中则证明问题已解
7. tar1=[tar[0]-1,tar[1]]#tar1为tar上面结点
8. tar2=[tar[0],tar[1]-1]#tar2为tar左边结点
9. **if** tar1 **in** CLOSED **or** tar2 **in** CLOSED:
10. CLOSED.append(tar)
11. **print**("走出迷宫的路径为:")
12. **for** i **in** CLOSED:
13. **print**(i)
14. **else**:
15. **print**("迷宫不可解!")

**三、A\*算法改进及优化**

**1.A\*算法改进及优化**

利用A\*算法解决迷宫问题的一般步骤如下：

1.建立迷宫的表示：与深度优先算法类似，可以使用二维数组或图数据结构来表示迷宫。同样，可以用0表示道路，1表示墙壁。

2.定义启发式函数：A\*算法需要一个启发式函数来评估当前点到终点的距离。对于迷宫问题，可以定义一个启发式函数h(n)=manhattan(start,end)-manhattan(n,end)，其中manhattan 函数计算两点之间的曼哈顿距离，start 和 end 分别为起点和终点，n 为当前点。

3.初始化开放列表和闭合列表：将起点加入开放列表，并将其他所有点加入闭合列表。

4.选择最佳节点：遍历开放列表，计算每个节点的启发式函数值，选择具有最小启发式函数值的节点作为当前最佳节点。

5.扩展节点：将最佳节点的上下左右四个相邻节点加入开放列表，并计算它们的启发式函数值。同时，将最佳节点从开放列表移除，加入闭合列表。

6.判断循环：重复步骤 4 和步骤 5，直到找到终点。此时，算法将输出从起点到终点的最短路径。

7. 路径回溯：根据开放列表中的节点记录，回溯到起点，还原出最短路径。

以下是一个简单的示例代码：

1. **import** tkinter as tk
2. **import** time
4. #创建迷宫
5. **def** create\_maze(draw):
6. #在这里修改迷宫,0是可走路径,1是墙体
7. maze=[
8. [0,1,1,0,1],
9. [0,1,0,0,0],
10. [0,0,0,1,0],
11. [1,1,0,1,0],
12. [0,0,0,1,0]
13. ]
15. #划分网格
16. **for** i **in** range(0,len(maze)\*40,40):
17. x0,y0=0,i
18. x1,y1=len(maze[0])\*40,i
19. draw.create\_line(x0,y0,x1,y1)
20. **for** j **in** range(0,len(maze[0]\*40),40):
21. x0,y0=j,0
22. x1,y1=j,len(maze)\*40
23. draw.create\_line(x0,y0,x1,y1)
25. **for** i **in** range(len(maze)):
26. **for** j **in** range(len(maze[0])):
27. x0,y0=j\*40,i\*40
28. x1,y1=x0+40,y0+40
29. **if** maze[i][j]==1:
30. draw.create\_rectangle(x0,y0,x1,y1,fill='black')
31. draw.update()
33. **return** maze
35. **class** mazeSolve:
36. **def** \_\_init\_\_(self,maze):
37. #迷宫数组
38. self.maze=maze
39. #OPEN和CLOSED表
40. self.OPEN=set()
41. self.CLOSED=set()
42. #迷宫的行数和列数
43. self.rows=len(self.maze)
44. self.cols=len(self.maze[0])
45. #迷宫的起点和终点
46. self.start=(0,0)
47. self.tar=(self.rows-1,self.cols-1)
48. #字典类型记录结点的前驱结点,键是结点,值是前驱
49. self.pre={}
51. **def** calculate\_g(self,cur):
52. #计算当前结点的实际代价g(n)
53. g\_cost=0#计算当前结点的代价时初始化g(n)=0
54. **while** cur **in** self.pre:
55. g\_cost=g\_cost+1
56. cur=self.pre[cur]
58. **return** g\_cost
60. **def** calculate\_h(self,cur):
61. #计算当前结点到目标结点的曼哈顿距离作为估计代价
62. **return** abs(cur[0]-self.tar[0])+abs(cur[1]-self.tar[1])
64. **def** calculate\_f(self,cur):
65. #估价函数
66. g\_cost=self.calculate\_g(cur)
67. h\_cost=self.calculate\_h(cur)
68. **return** g\_cost+h\_cost
70. **def** getNode(self):
71. #通过估价函数找到OPEN表估价值f(n)最小的结点进行拓展
73. #初始设最小估价值为无穷大
74. min\_f=float('inf')
75. #该结点初始设为空
76. cur=None
77. **for** i **in** self.OPEN:
78. **if** self.calculate\_f(i)<min\_f:
79. min\_f=self.calculate\_f(i)
80. cur=i
81. **return** cur
83. **def** wideNodes(self,cur):
84. #拓展结点,只要不超出迷宫范围,每个方向上都进行拓展
85. #后序通过isPath()函数判断是否为可走路径
86. row=cur[0]
87. col=cur[1]
88. #拓展结点列表
89. nodes=[]
90. **if** row>0:
91. #向左
92. nodes.append((row-1,col))
93. **if** col>0:
94. #向上
95. nodes.append((row,col-1))
96. **if** row<self.rows-1:
97. #向下
98. nodes.append((row+1,col))
99. **if** col<self.cols-1:
100. #向右
101. nodes.append((row,col+1))
102. **return** nodes
104. **def** isPath(self,node):
105. #判断node结点是否为可走路径
106. row=node[0]
107. col=node[1]
108. **return** self.maze[row][col]==0
110. **def** finalPath(self):
111. #从终点开始不断寻找其前驱结点,直到找到起点
112. cur=self.tar
113. path=[cur]
114. **while** cur!=self.start:
115. #cur更新为其前驱结点
116. cur=self.pre[cur]
117. path.append(cur)
119. path.reverse()
120. **return** path

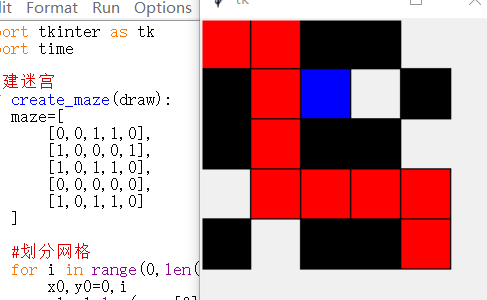
**2．时间复杂度分析**

算法的时间复杂性即时间复杂度，指算法执行所需的时间，要衡量算法的执行时间，一个方法是做基准分析，即记录程序计算出结果所消耗的实际时间。但基准分析计算的是执行算法的实际时间，它依赖于特性的计算机、程序、时间、编译器与编程语言。因此摆脱程序或计算机的影响描述算法的时间复杂性时，量化算法的操作或步骤很重要，这样的衡量方法称为数量级方法，常被称作大0记法，记作O(f(0))。这里参数1n指问题规模，T(n)是当问题规模为n时，解决问题所需的时间是T(n),f(n)为T(n)函数中起决定性作用的部分提供了简单的表示。

A\*算法的时间复杂度可以从前文对A\*算法运行步骤分析得出。首先创建两个空表，将起始节点放入闭合列表中，并将起始节点的所有相邻节点放入开放列表中。判断开放列表是否为空，如果为空，则寻路失败；如果不为空，则选取z最小的节点判断其是否为目标节点，如果是，则寻路成功。此时时间复杂度为常数阶。不过讨论一个算法的时间复杂度，往往以最坏的情况为标准，因为无论状态空间多大，多复杂，都不可能超过这个时间。所以A\*算法在寻路过程中，有可能将地图上所有的节点都考察一遍，才能找到目标节点，此时的时间复杂度为线性阶。算法在搜索过程中，所涉及到的节点都 需要使用到估价函数，这个函数是前面提到的曼哈顿距离，在整个算法过程中，由于只是进行加减运算，对时间复杂度的影响不大。在问题规模为n的寻路地图中，A\*算法的时间复杂度为O(n)。

**五、A\*算法的可视化思维模型**

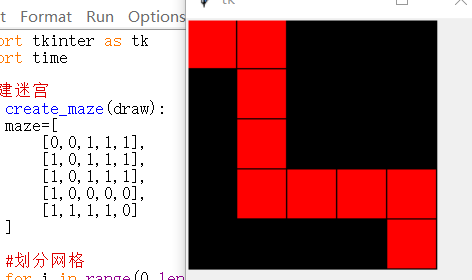
在python环境下，构造一个真实的有弯路的环境模型，在此环境中，运行标准A\* 算法，对有弯路的模型进行路径搜索。如下图所示，图中的每一个方格，代表一个节点，两个方格之间的边没有表示出来，默认两个方格是相邻的。黑色区域表示不能通过，其余空白区域代表可以通过的路径，蓝色区域表示测试节点，红色区域代表最终路径。图中(1,1)代表开始，（5,5）代表结束,通过使用A\*算法搜索到一条从开始到结束的最短通路。



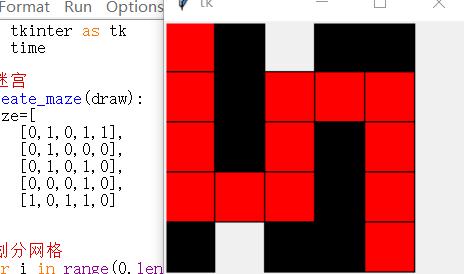
以下是一个简单的示例代码：

1. **def** seekPath(self,draw):
2. #draw参数,是tkinter.Tk()对象,用于可视化
4. #起点放入OPEN表中
5. self.OPEN.add(self.start)
7. **while** self.OPEN:#OPEN不为空就进行搜索
8. #寻找OPEN表中f(n)值最小的作为拓展结点
9. cur=self.getNode()
10. **if** cur==self.tar:
11. #如果f(n)最小的结点为目标结点,则通过前驱结点返回该路径,并进行可视化
12. finalPath=self.finalPath()
13. #可视化最终路径
14. self.draw\_finalPath(draw,finalPath)
16. #cur移出OPEN表,移入CLOSED表
17. self.OPEN.remove(cur)
18. self.CLOSED.add(cur)
20. **for** i **in** self.wideNodes(cur):
21. #当拓展结点已经在CLOSE表中或者它不为可走路径时遍历其他拓展结点
22. **if** i **in** self.CLOSED **or** **not** self.isPath(i):
23. **continue**
25. #当拓展结点不在OPEN表中时将其放入OPEN表中
26. **if** i **not** **in** self.OPEN:
27. self.OPEN.add(i)
29. #设置拓展结点的前驱结点为父节点
30. self.pre[i]=cur
31. self.draw\_searchPath(draw,cur)
32. draw.update()
33. #每一秒中移动一次
34. time.sleep(1)
36. **def** draw\_searchPath(self,draw,cur):
37. #可视化搜索路径,用蓝色标记
38. row=cur[0]
39. col=cur[1]
41. x0,y0=col\*40,row\*40
42. x1,y1=x0+40,y0+40
44. draw.create\_rectangle(x0,y0,x1,y1,fill='blue')
46. **def** draw\_finalPath(self,draw,path):
47. **for** i **in** path:
48. row=i[0]
49. col=i[1]
50. x0,y0=col\*40,row\*40
51. x1,y1=x0+40,y0+40
52. draw.create\_rectangle(x0,y0,x1,y1,fill='red')
54. #设置一个函数方便可视化
55. **def** solve\_maze():
56. #删除前一次所画全部内容
57. draw.delete('all')
58. maze=create\_maze(draw)
59. solve=mazeSolve(maze)
60. solve.seekPath(draw)
62. #创建Tk对象
63. A\_star=tk.Tk()
64. draw=tk.Canvas(A\_star,height=6\*40,width=6\*40)
65. draw.pack()
67. #设置一个交互按钮,按下此按钮便可调用solve\_maze()进行寻路
68. button=tk.Button(text='开始寻路',command=solve\_maze)
69. button.pack()

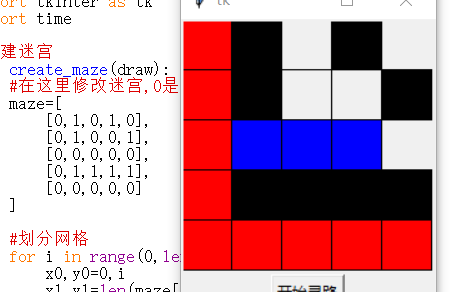
**仿真测试结果如下：**

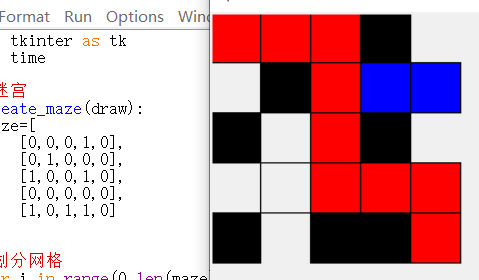
****

上图中生成的迷宫只有一条路径，即所有可行考察节点即为最短路径节点。

****

上图中生成的迷宫包含多个死胡同，图中可以看出，通往死胡同的节点并没有被考察，从而说明A\*算法简化了搜素时间。

****

****

以上生成的两个迷宫均有被考察但没列入最终路径当中，从而说明即便有其他路径可以到达终点，但在启发式函数的作用下能及时回到最短路径的方向上，从而避免了对多重路径求最优路径的计算。

**结论：**

深度优先搜索算法是相对比较简单的算法，实现容易，但是算法中没有描述考虑到相邻节点重复判定问题，即多个节点存在相同的相邻节点，系统如果在第二次访问同一节点时，很容易进入循环，寻路算法无法终止。而A算法通过使用估价函数来评估每个节点的价值，从而避免了在无益的搜索路径上浪费时间。

虽然A\*算法优势比较明显，而且寻路过程中可以更智能地分析最优路径并减少搜索的冗余节点，但是标准A\*算法也存在不足:

1.标准A\*算法通过开放列表和闭合列表存放节点的数据信息，当找到目标节点时，需要根据闭合列表存放的信息回溯，进而找到一条路径。但是当问题空间比较复杂，有一些无法通过的路径或者中断的路径时，A\*算法在根据搜索函数计算会考虑这些节点，进而在回溯时会重复考察。虽然最后会排除这些多余节点，但是也会浪费很多时间。

2.标准A\*算法在解决多个起始节点和目标节点的复杂寻路问题时，常规思路是多次调用A\*算法，分别求解每个初始点到每个终止点的路径，通过比较，最后得到一条(或多条)最优路径。如此一来，计算量呈几何式增长。

**参考文献：**

1.张永旭 基于路径搜索的改进A\*算法研究[J] 2018.5

2.刘超 人工智能教学中的深度优先算法——以经典迷宫问题为例[J] 2021.9