人工智能实验

深度优先搜索与迭代加深搜索解决迷宫问题

16 级计科教务 2 班

16337327

郑映雪

实验题目

深度优先搜索与迭代加深搜索解决迷宫问题

实验内容

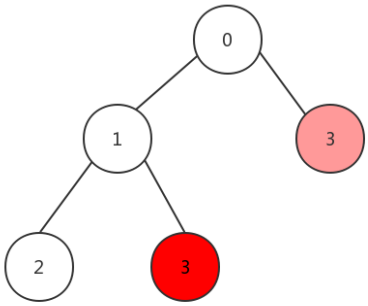
算法原理

1、深度优先搜索（DFS）

深度优先搜索从起点开始，选择一个子节点进行访问，并在子节点的子节点中选择一个进行访问……如此反复下去，直到该子节点为叶节点或该节点为目标节点，则进行回溯。由于这种搜索“尽量往深处搜索”的特性，所以名为深度优先搜索。

在本次实验的实例中，从起点开始，子节点为其上、下、左、右四个节点。当到达边界时、前路为墙时或该点已经访问过时则不访问该节点而回溯，否则访问该节点并搜索它的子节点，直到可以到达终点坐标。

注意到这个迷宫到达终点本身就有很多条路，所以在这个实验里我输出了所有可能的路径。因为深度优先搜索是不具备最优性的，如下图所示：



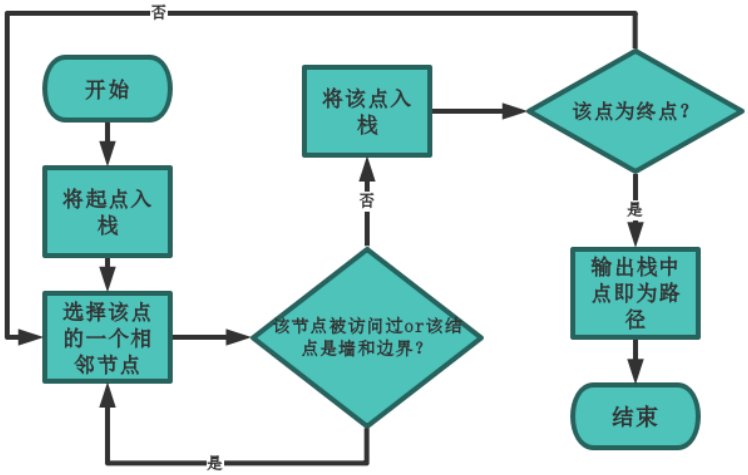
按照深度优先的算法，在这里会先搜索到红色的目标，但是最优路径应该是粉红色的目标才对。于是有了迭代加深搜索。

2、迭代加深搜索

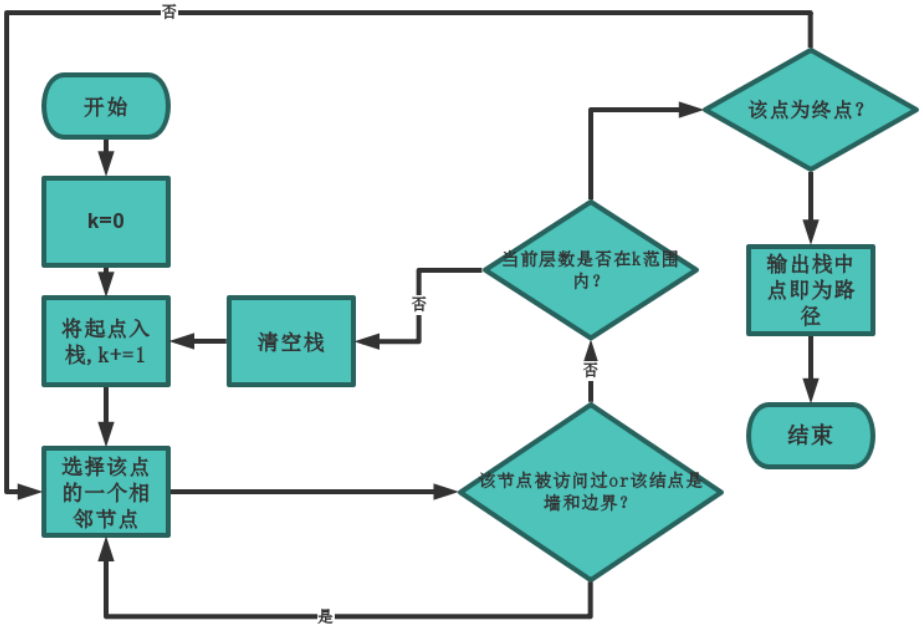
迭代加深搜索在深度优先搜索的基础上进行了变形。它一次次限制可以搜索到的最多层数，逐层叠加，这样就可以保证自己找到目标节点的第一个路径是最优路径。

流程图

深度优先搜索：



迭代加深搜索：



关键代码

深度优先搜索：

1、数据结构与准备工作

设置结点的结构，包括该节点的横、纵坐标，同时做好相关准备工作。

```
class node:#结点结构
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
stack = []#存放路径走过的结点
stacks = []#存放所有可达到目标的路径
minstep = 10000#判断最小路径的变量
length=[]
index = 0
bestindex = 0#判断最小路径的下标
visit = []#该点是否访问过
behave = [[0, 1], [0, -1], [-1, 0], [1, 0]]  # 上下左右操作
data = []#数据集
```

2、DFS 函数

①不满足继续深度探索的条件

不需要继续探索的条件有两种：一种是越界和碰墙，另一种是到达终点。到达终点后，将路径中存储的结点的 x、y 值都形象化地表示出来，显示出路径和迷宫。越界时，则不访问当前节点。

```
if x < 1 or x > 16 or y < 1 or y > 34:  # 越界
    return
if x == 16 and y == 2:#到达终点
    print('step:', len(stack))
    outdata = []#输出迷宫及路线
    for datas in data:
        tmp = []
        for datass in datas:
            tmp.append(datass)
        outdata.append(tmp)
    for j in range(len(stack)):
        outdata[stack[j].x][stack[j].y] = '*'
    for out1 in outdata:
```

```
        string = ''
        for out2 in out1:
            if out2 == 1:
                string += '| '
            elif out2 == 0:
                string += '  '
            elif out2 == '*':
                string += '. '
            else:
                string += 'E '
        print(string)
    print('\n---\n')
    stacks.append(outdata)#当前可到达目标的路径存储起来
    length.append(len(stack))
    index += 1
    if minstep > len(stack):#判断是否是最短路径的结果
        minstep = len(stack)
        bestindex = index
    return
```

②继续深度探索的条件

对当前节点进行上下左右的操作。如果满足继续探索的条件，则将当前节点标记为已访问，并将当前节点入访问栈。然后递归访问下一节点。如若回溯，则将回溯路径上的结点重新标记为未访问，并且删除栈中回溯路径上的结点。

```
for i in range(4):#上下左右操作
    newx = x + behave[i][0]
    newy = y + behave[i][1]
    if newx >= 1 and newx <= 16 and newy >= 1 and newy <= 34 and
data[newx][newy] == 0 and visit[newx][
        newy] == 0 and data[newx][newy] != 8:
        visit[newx][newy] = 1
        current = node(newx, newy)
        stack.append(current)#满足条件，结点入栈，标记为访问过
        dfs(newx, newy)#递归访问下一节点
        visit[newx][newy] = 0#回溯时将回溯路径上结点标记为未访问
        del (stack[-1])#删除栈中回溯路径上的结点
```

3、输出部分

在 DFS 函数里我们输出了深度优先搜索所有搜到的路径，最终需要使用 minstep 和 bestindex 来输出最优路径。

```
for j in range(len(length)):
    if length[j]==minstep:
        for out1 in stacks[j]:
            string = ""
            for out2 in out1:
                if out2 == 1:
                    string += '| '
                elif out2 == 0:
                    string += '  '
                elif out2 == '*':
                    string += '. '
                else:
                    string += 'E '
            print(string)
print('the minstep is:', minstep)
```

迭代加深搜索

1、相关准备

迭代加深搜索相对于 DFS 多了探索的最大层数 K，所以我们设置限制层数的相关变量。

```
class node:#结点结构
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
stack = []#存放路径走过的结点
visit = []#该点是否访问过
behave = [[0, 1], [0, -1], [-1, 0], [1, 0]]  # 上下左右操作
data = []#数据集
k = 0#允许探寻的最大层数
tmpk = 0#探索时的层数
flag = 0#是否到达终点
```

2、迭代加深的 DFS 函数

①不满足继续深度探索的条件

与 DFS 相似，不需要继续探索的条件有三种：一种是越界和碰墙，一种是到达终点，多了的一种是当前探索的层数已经到达了 K 值。到达终点后，将路径中存储的结点的 x、y 值都形象化地表示出来，显示出路径和迷宫，且根据迭代加深的性质，此时的解已经是最优解。越界时或超过 K 层时，则不访问当前节点。

```
global flag, tmpk, k
if x < 1 or x > 16 or y < 1 or y > 34 or tmpk > k:  # 越界
    return
if x == 16 and y == 2 and tmpk <= k:#到达终点且在当前可以探测的层数之内，则输出当前解，当前解就是最优解
    print('step:', len(stack))
    outdata = []
    for datas in data:
        tmp = []
        for datass in datas:
            tmp.append(datass)
        outdata.append(tmp)
    for j in range(len(stack)):
        outdata[stack[j].x][stack[j].y] = '*'
    for out1 in outdata:
        string = ''
        for out2 in out1:
            if out2 == 1:
                string += '| '
            elif out2 == 0:
                string += '  '
            elif out2 == '*':
                string += '. '
            else:
                string += 'E '
        print(string)
    flag = 1
    return
```

②继续深度探索的条件

对当前节点进行上下左右的操作。如果满足继续探索的条件（结点可走、未达 K 值），则将当前节点标记为已访问，并将当前节点入访问栈。然后递归访问下一节点。如

若回溯，则将回溯路径上的结点重新标记为未访问，并且删除栈中回溯路径上的结点，同时也要将当前已经探索的层数减 1。

```
for i in range(4):#上下左右操作
    newx = x + behave[i][0]
    newy = y + behave[i][1]
    if newx >= 1 and newx <= 16 and newy >= 1 and newy <= 34 and
data[newx][newy] == 0 and visit[newx][
        newy] == 0 and data[newx][newy] != 8:
        visit[newx][newy] = 1
        current = node(newx, newy)
        stack.append(current)#满足条件，结点入栈，标记为访问过
        tmpk += 1#层数加 1
        dfs(newx, newy)#递归访问下一节点
        visit[newx][newy] = 0#回溯时将回溯路径上结点标记为未访问
        del (stack[-1])#删除栈中回溯路径上的结点
        tmpk -= 1#层数减 1
```

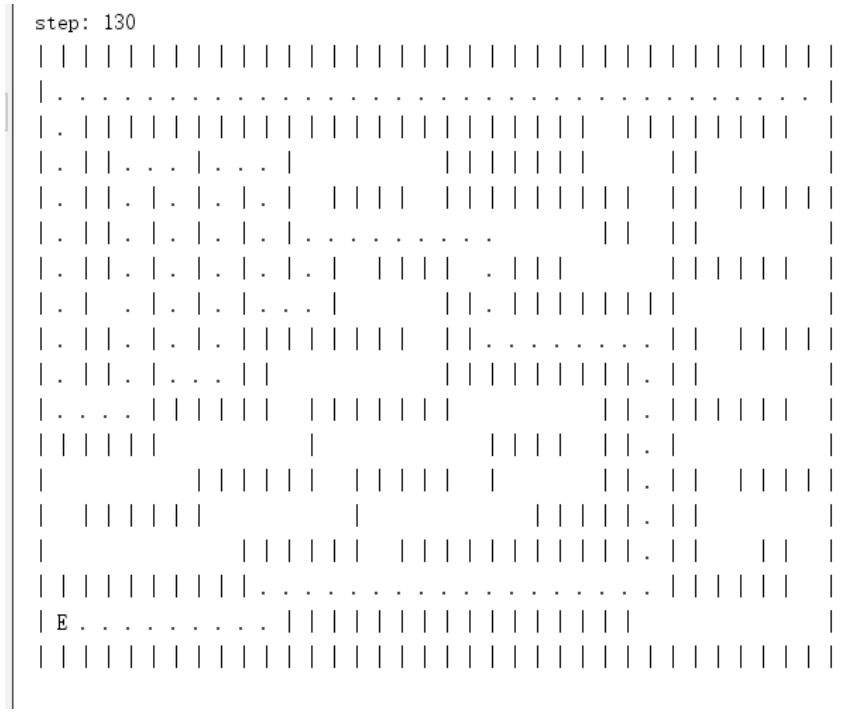
实验结果及分析

实验结果展示

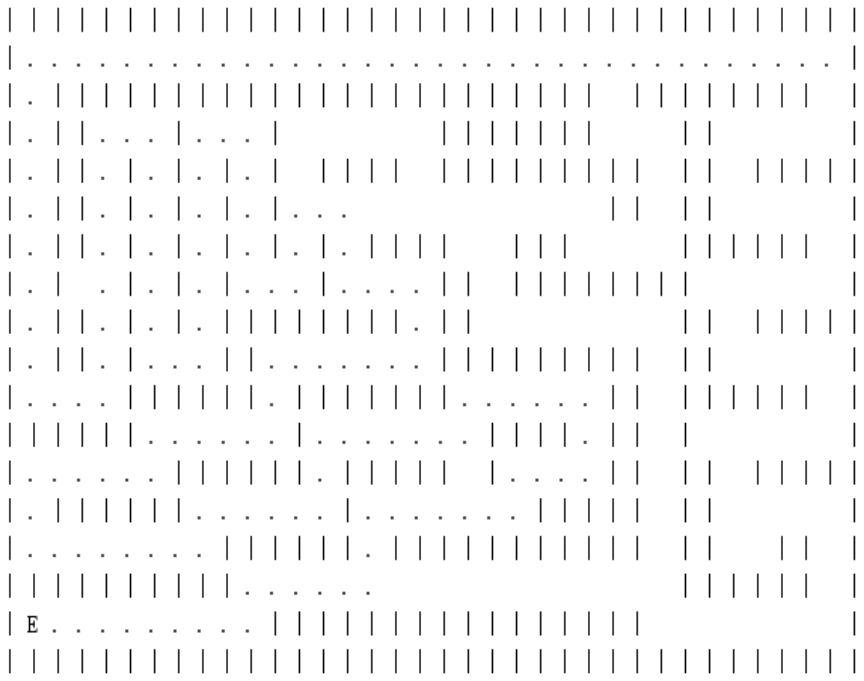
1、深度优先搜索

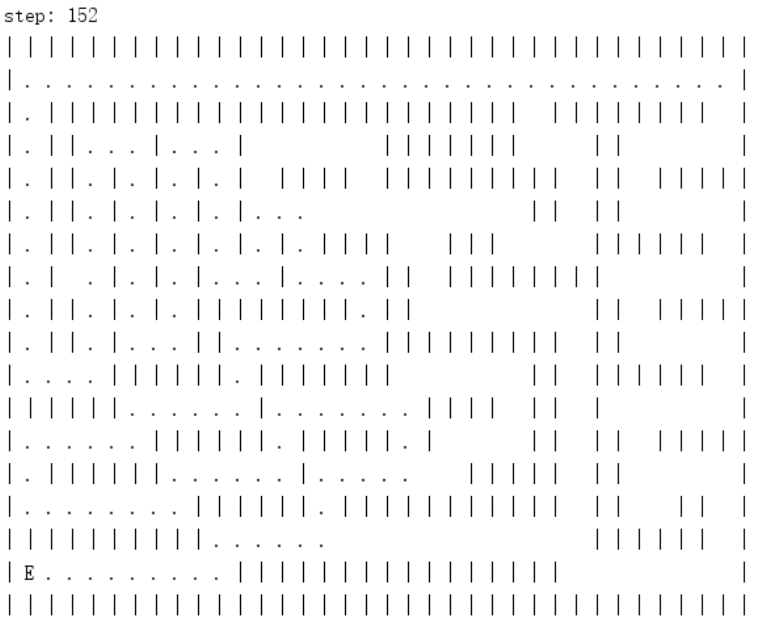
深度优先搜索的输出是所有满足条件的解，通过比较步长选择最优解。由于输出太多，此处只放上几张图。

step: 130



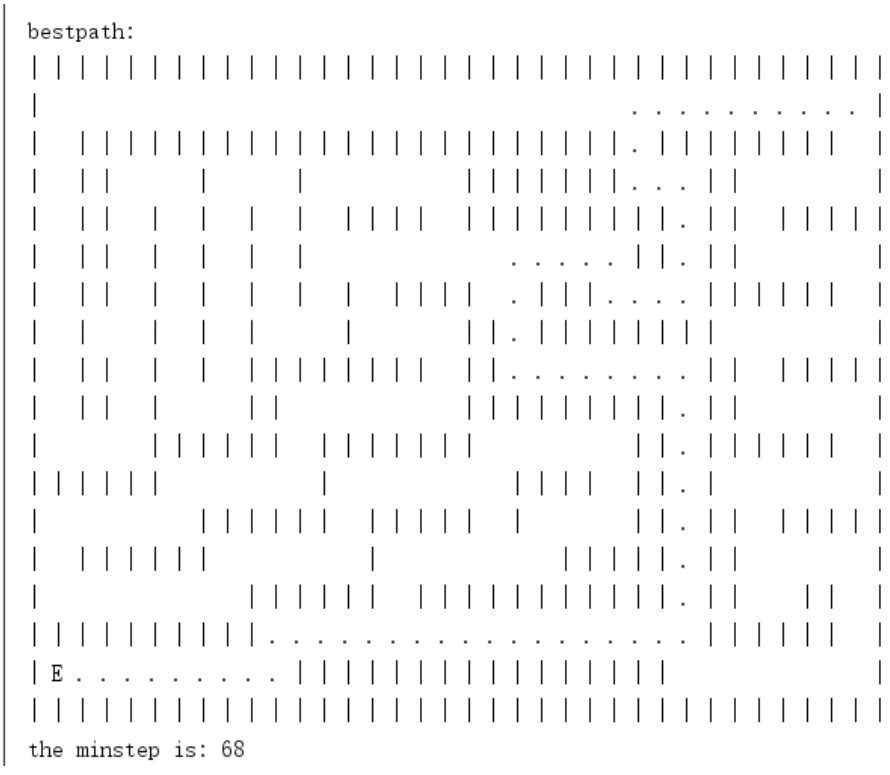
step: 164





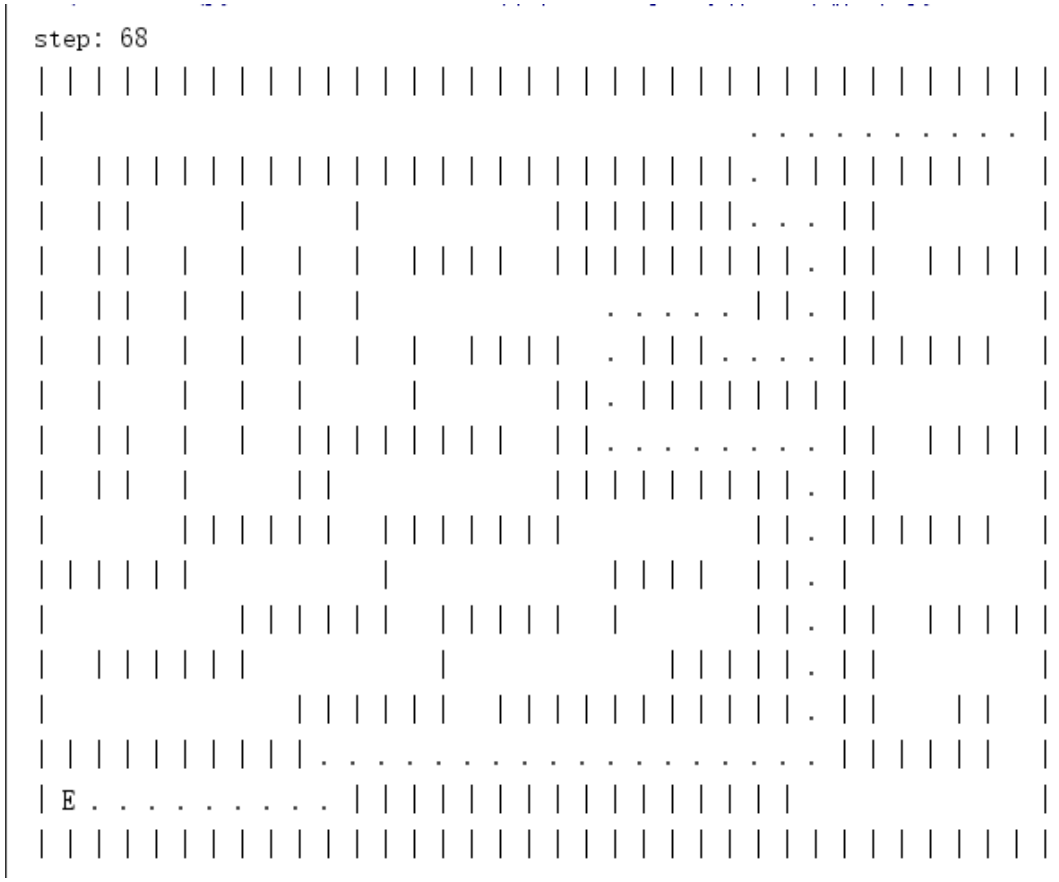
.....

输出所有路径之后通过比较输出最优路径和最小步长：



迭代加深搜索

迭代加深搜索本身的性质决定它搜索到了第一个解即为最优解。



可以看出来的此处的第一个解即为深度优先搜索里的的最优解。

评测指标展示

时间比较

深度优先搜索：

time: 0.0838590926603614

迭代加深搜索：

time: 0.059936349948138734

由此可知，寻找到解时间，深度优先搜索是比迭代加深搜索要慢一些的。因为本实验中的数据较小，所以两者差别并不是非常大，有些特殊的数据，迭代加深搜索作为模拟宽度优先搜索的存在，是可以比深度优先搜索快很多的。

思考题——方法比较

时间复杂度和空间复杂度

设 b 为每个节点的的子节点最多数目，深度优先搜索的时间复杂度是$O(b^m)$,其中 m 为搜索树的最深层数；迭代加深搜索的时间复杂度为$O(b^d)$,其中 d 为最优解所在的层数。

深度优先搜索的空间复杂度为 O（bm），迭代加深搜索的的空间复杂度为$O(b^d)$。

其他的搜索方法：

宽度优先搜索 BFS，时间复杂度为$O(b^d)$，空间复杂度也为$O(b^d)$

一致代价搜索，在宽度优先搜索的基础上优先选择代价最小的路径首先搜索，时间复杂度和空间复杂度都为$O(b^{1+\left[\frac{C^*}{\epsilon}\right]})$，相比宽度优先搜索来说在时间和空间上进行了节省。

双向搜索在宽度优先搜索的基础上进行两边同时搜索，时间和空间复杂度均为$O(b^{\frac{d}{2}})$。大大节省了宽度优先搜索的时间和空间。

优缺点比较及适用场景

深度优先搜索的优点在于它的空间复杂度很低，对于大数据来说不用占用很多的内存。但是深度优先搜索不具有完备性（除非完全遍历完所有解）和最优性。深度优先第一次搜索到的解在很多情况下不会是最优解。它适用在只找到解即可但不强求最优解、同时追求节省空间的情况下。

迭代加深搜索是利用了受限的深度优先搜索来模拟宽度优先搜索，从而得到宽度优先搜索的完备性和最优性。它本质上又还是深度优先搜索，所以空间复杂度比宽度优先搜索低，但一层一层地逐层搜索比单纯的深度优先搜索时间复杂度要高一些。但综合来看迭代加深搜索是一个效率相对比较高的的搜索。它适用于节省空间又追求最优解的场景。

其他搜索方法：

宽度优先搜索相比深度优先搜索来说具有了最优性，即它首先找到的解肯定是最优解。但是宽度优先搜索需要存储搜索树的所有边界结点，这将耗费大量的空间。它适用于需要找到最优解、不在乎空间浪费的场景。

一致代价搜索相比宽度优先搜索来说更加好一些，它每次都从代价最小的节点先开始扩展。相比起来，一致代价搜索可以一定程度上减少时间和空间的复杂度，多了的是记录当前所有边界节点的代价。

双向搜索是起点和终点分别进行宽度优先搜索，在保证在最优性的情况下，时间和空间复杂度都进行了开平方的降维优化。但是双向搜索实现较为复杂，同时也不一定有解。