中山大學
SUN YAT-SEN UNIVERSITY

《人工智能实验》

实验报告

Lab 9

无信息搜索 & 启发式搜索

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	17 级计算机科学与技术
学 生 姓 名 ：	薛伟豪
学 号 ：	17341178
联 系 方 式 ：	15013041671

Lab 9：无信息搜索 & 启发式搜索

1. 算法原理

1.1. 深度优先搜索（DFS）

对于一个给定的无向连通图，DFS首先选择一个节点v_1作为起始节点，然后从v_1出发，访问与v_1相邻且未被访问过的任意一个节点v_2，再访问与v_2相邻且未被访问过的任意一个节点v_3，…，以此类推，直到当前访问的节点v_k没有与其相邻且未被访问过的节点。此时，DFS会回退到最近被访问过的节点v_{k-1}，再次对与v_{k-1}相邻的节点进行判断。如果不存在未被访问过的节点，则继续回退；如果存在，则访问该节点，并重复上述过程，直到图中所有的节点都被访问过。

1.2. 宽度优先搜索（BFS）

对于一个给定的无向连通图，BFS首先选择一个节点v1作为起始节点，然后从v_1出发，依次访问所有与v1相邻且未被访问过的节点v_{11}, v_{11}, …, v_{1k}，然后再依次访问所有与v_{11}相邻且未被访问过的节点，依次访问所有与v_{12}相邻且未被访问过的节点，…，以此类推，直到图中所有的节点都被访问过。

1.3. 一致代价搜索

一致代价搜索基于宽度优先搜索。与宽度优先搜索不同的是，一致代价搜索会维持一个优先级队列，访问该节点的成本越低，则该节点优先级越高。这里的成本可以是该节点到起始节点的距离等。在算法运行过程中，每次让优先级最高的节点出队列并对其进行访问。如果所有节点的访问成本均相同，则一致代价搜索会退化成宽度优先搜索。

1.4. 迭代加深搜索

迭代加深搜索基于深度优先搜索。与深度优先搜索不同的是，迭代加深搜索对深度优先搜索的递归层数进行了限制。迭代加深搜索的一般步骤如下：

1) 首先，设置深度限制L=0，迭代地增加深度限制L，对于每个深度限制都进行深度受限

搜索。

2) 如果在限定的深度限制L内，我们找到了目标节点，则结束搜索。如果没有未访问的节点可以扩展，则提高深度限制L。
3) 如果深度限制L不能再提高（即达到最大层数），且仍未找到目标节点，那么说明图中所有节点都被访问过，没有答案，此时结束搜索。

1.5. A*搜索

A* 搜索算法是启发式搜索算法，它在BFS算法的基础上加入了启发式信息。启发式搜索算法的关键在于，从当前节点选择下一步要搜索的节点时，会通过一个估值函数选择代价最小的节点。相较于BFS的“盲目性”，A*搜索可以有效地减少搜索时间，得到一个较优解。启发式搜索算法的核心在于估值函数的设计：

$$f(n) = g(n) + h(n)$$

其中$g(n)$为起点到当前节点的代价，$h(n)$为启发式函数，有多种选择。

A*算法描述如下：从起始节点开始，不断地查询当前节点周围可访问的节点，并计算它们的$g(n)$，$h(n)$得到$f(n)$。一直选择估值函数$f(n)$最小的节点进行下一步的扩展，同时更新已经被访问过的节点的$g(n)$，直到找到目标节点。

1.6. IDA*搜索

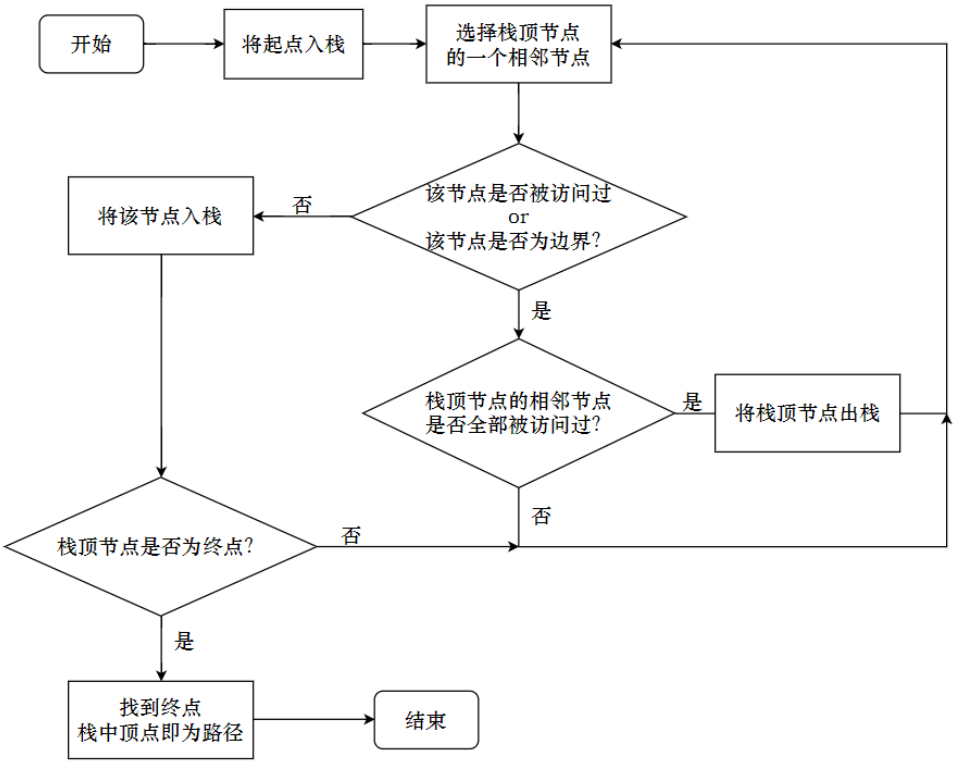
IDA* 搜索算法是迭代加深深度优先搜索算法（IDS）的一个扩展。由于不用维护列表，所有其空间复杂度远小于A* 搜索算法，在搜索图为有向图的时候，它的性能会比A*更好。

具体地，IDA*算法步骤如下：

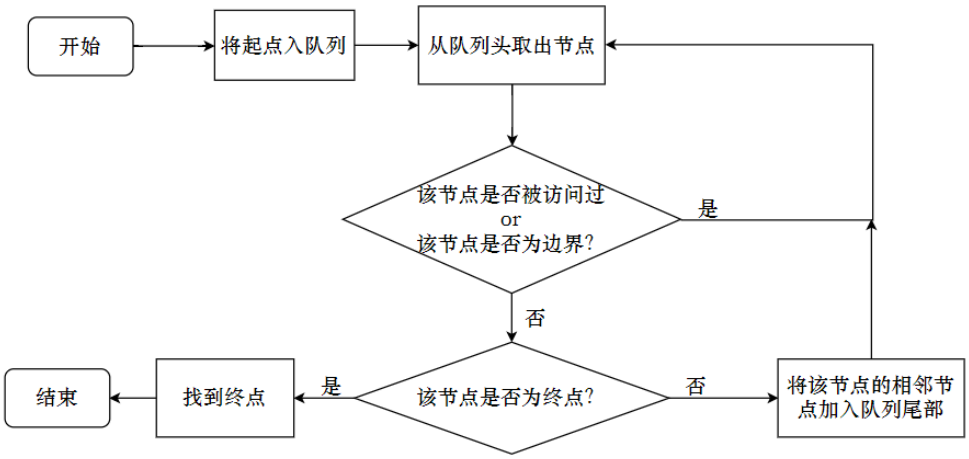
1) 将阈值设为起始节点的$h(n)$值。
2) 从起始节点开始进行深度受限搜索，搜索的范围为估值函数不超过阈值的节点。
3) 如果没有找到目标节点，则将阈值设为步骤2中已经探知的，估值函数超过阈值的所有节点的最小值，再次进行步骤2。如果找到目标节点，则结束搜索。

2. 流程图

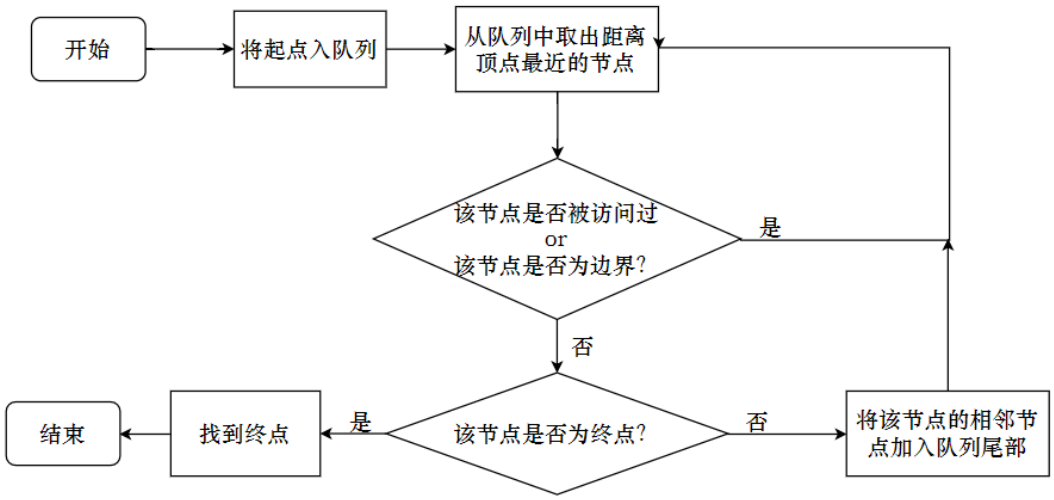
2.1. 深度优先搜索（DFS）



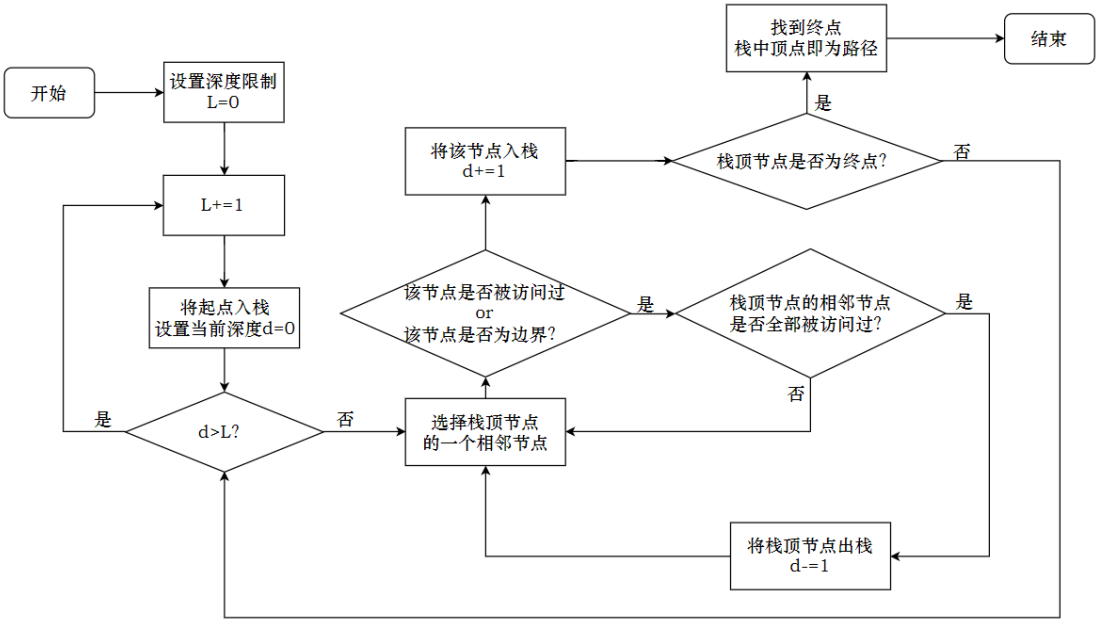
2.2. 宽度优先搜索（BFS）



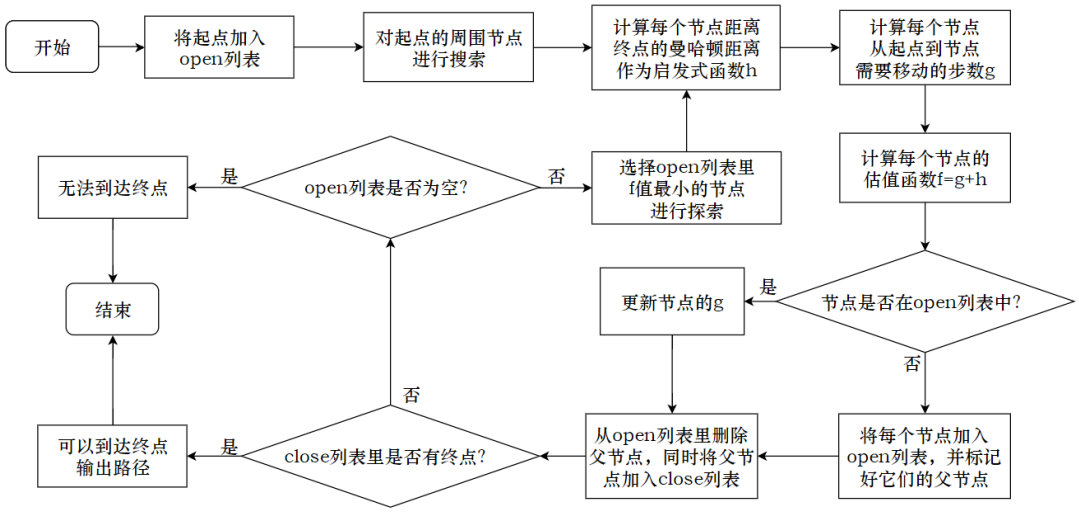
2.3. 一致代价搜索



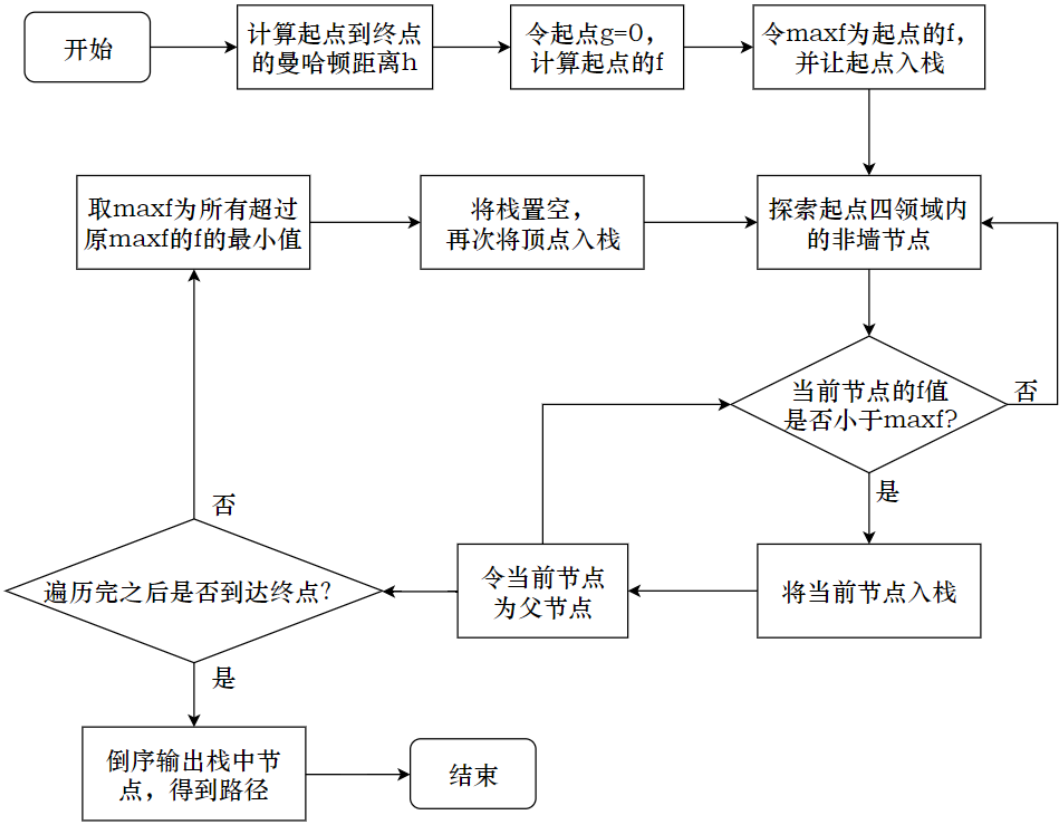
2.4. 迭代加深搜索



2.5. A*搜索



2.6. IDA*搜索



3. 重要代码展示

- **深度优先搜索**

```
class DFS:
    def search(self, pos, pre_pos):
        '''
        对位置 pos 进行搜索，pre_pos 为该位置的上一个位置，depth 为当前深度
        参数说明：
        maze：用于存储迷宫图
        is_visited：用于记录迷宫的位置是否被访问过
        pre：用于记录每个位置上一步的位置信息
        reached：标记是否到达终点
        '''
        x = pos[0]
        y = pos[1]
        #如果当前位置是墙壁，则结束搜索
        if maze[x][y] == '1':
            return
        #如果当前位置是终点，则更新对应参数，结束搜索
        if self.maze[x][y] == 'E':
            self.reached = True
            self.pre[x][y] = pre_pos
            return
        #如果当前位置已经被访问过，结束搜索
        if self.is_visited[x][y] == True:
            return
        #将当前位置标记为已被访问，同时更新当前位置的上一步位置信息
        self.is_visited[x][y] = True
        self.pre[x][y] = pre_pos
        #分别沿左、上、右、下进行递归搜索
        if x-1>=0 and self.reached==False:
            self.search((x-1, y), pos)
        if y-1>=0 and self.reached==False:
            self.search((x, y-1), pos)
        if x+1<len(self.maze) and self.reached==False:
            self.search((x+1, y), pos)
        if y+1<len(self.maze[0]) and self.reached==False:
            self.search((x, y+1), pos)
        #回溯时将当前位置重新标记为未被访问
        self.is_visited[x][y] = False
```

- **迭代加深搜索**

```
class IDS:
    def search(self, pos, pre_pos, depth):
        '''
        对位置 pos 进行搜索，pre_pos 为该位置的上一个位置，depth 为当前深度
        参数说明：
        maze：用于存储迷宫图
        is_visited：用于记录迷宫的位置是否被访问过
        pre：用于记录每个位置上一步的位置信息
        limit：当前的深度限制，初始化为 0
        reached：标记是否到达终点
        '''
        x = pos[0]
        y = pos[1]
        #如果当前的深度超过的最大深度限制，结束搜索
        if depth > self.limit:
            return
        #如果当前位置是墙壁，则结束搜索
        if maze[x][y] == '1':
            return
        #如果当前位置是终点，则更新对应参数，结束搜索
        if self.maze[x][y] == 'E':
            self.reached = True
            self.pre[x][y] = pre_pos
            return
        #如果当前位置已经被访问过，结束搜索
        if self.is_visited[x][y] == True:
            return
        #将当前位置标记为已被访问，同时更新当前位置的上一步位置信息
        self.is_visited[x][y] = True
        self.pre[x][y] = pre_pos
        #分别沿左、上、右、下进行递归搜索
        if x-1>=0 and self.reached==False:
            self.search((x-1, y), pos, depth+1)
        if y-1>=0 and self.reached==False:
            self.search((x, y-1), pos, depth+1)
        if x+1<len(self.maze) and self.reached==False:
            self.search((x+1, y), pos, depth+1)
        if y+1<len(self.maze[0]) and self.reached==False:
            self.search((x, y+1), pos, depth+1)
        #回溯时将当前位置重新标记为未被访问
        self.is_visited[x][y] = False
```

```
    def get_path(self, start, end):
        while not self.reached:
            # 未到达目的地，则增加搜索深度
            self.limit += 1
            # 清除上次搜索的记录
            self.pre = np.array([[None for i in range(len(maze[0]))]
for j in range(len(maze))])
            # 进行深度受限搜索
            self.search(start, None, 0)
        ……
```

- **IDA*搜索**

```
class IDAstar:
        '''
        参数说明：
        maze：用于存储迷宫图    start_node：起点    end_node：终点
        visited：存储已访问过的所有节点
        f：存储所有下一步可访问的，超过限定的节点的 f 值
        reached：标记是否到达终点
        last_node：标记最后一个访问的节点（若有解则为终点）
        '''
    def search_node(self, pos, depth):
        # 更新新节点的 g 和 h 值
        node = Node(pos, self.end_node.pos)
        node.g = depth
        return node

    def subsearch(self, node, pre_node, depth, limit):
        #节点的 f 值大于阈值，结束搜索，将该 f 值加入 f 列表
        if node.g + node.h > limit:
            self.f.append(node.g + node.h)
            return
        #如果当前位置是墙壁，则结束搜索
        if self.maze[node.pos[0]][node.pos[1]] == '1':
            return
        #如果当前位置是终点，则更新对应参数，结束搜索
        if node.pos == self.end_node.pos:
            node.father = pre_node
            self.reached = True
            self.last_node = node
            return
        #记录父节点
```

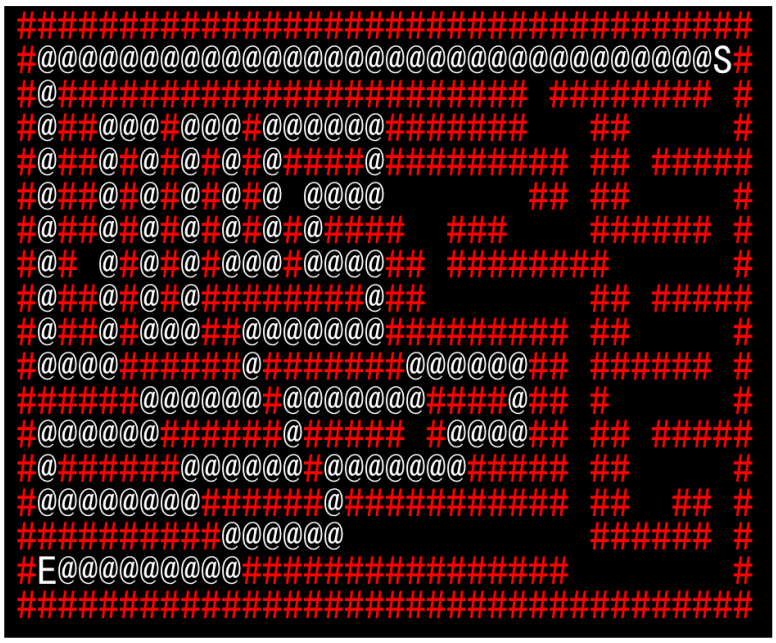
```
        node.father = pre_node
        #将节点加入已访问的列表
        self.visited.append(node.pos)
        x = node.pos[0]
        y = node.pos[1]
        #递归搜索所有相邻节点
        if x+1<len(self.maze) and not self.reached:
            next_pos = (x+1, y)
            #如果节点不在已访问的列表中，更新新节点的 g 和 h 值
            if next_pos not in self.visited:
                next_node = self.search_node(next_pos, depth)
                self.subsearch(next_node, node, depth + 1, limit)
        if x-1>=0 and not self.reached:
            next_pos = (x-1, y)
            #如果节点不在已访问的列表中，更新新节点的 g 和 h 值
            if next_pos not in self.visited:
                next_node = self.search_node(next_pos, depth)
                self.subsearch(next_node, node, depth + 1, limit)
        if y+1<len(self.maze[0]) and not self.reached:
            next_pos = (x, y+1)
            #如果节点不在已访问的列表中，更新新节点的 g 和 h 值
            if next_pos not in self.visited:
                next_node = self.search_node(next_pos, depth)
                self.subsearch(next_node, node, depth + 1, limit)
        if y - 1 >= 0 and not self.reached:
            next_pos = (x, y-1)
            #如果节点不在已访问的列表中，更新新节点的 g 和 h 值
            if next_pos not in self.visited:
                next_node = self.search_node(next_pos, depth)
                self.subsearch(next_node, node, depth + 1, limit)
        #回溯时将节点从已访问列表中删除
        self.visited.remove(node.pos)

    def search(self):
        # 阈值设为初始节点的 h 值
        limit = self.start_node.h
        while not self.reached:
            #如果没有找到目标节点，将阈值设为 f 列表中最小值
            self.subsearch(self.start_node, None, 1, limit)
            limit = min(self.f)
            self.f = []
        ……
```

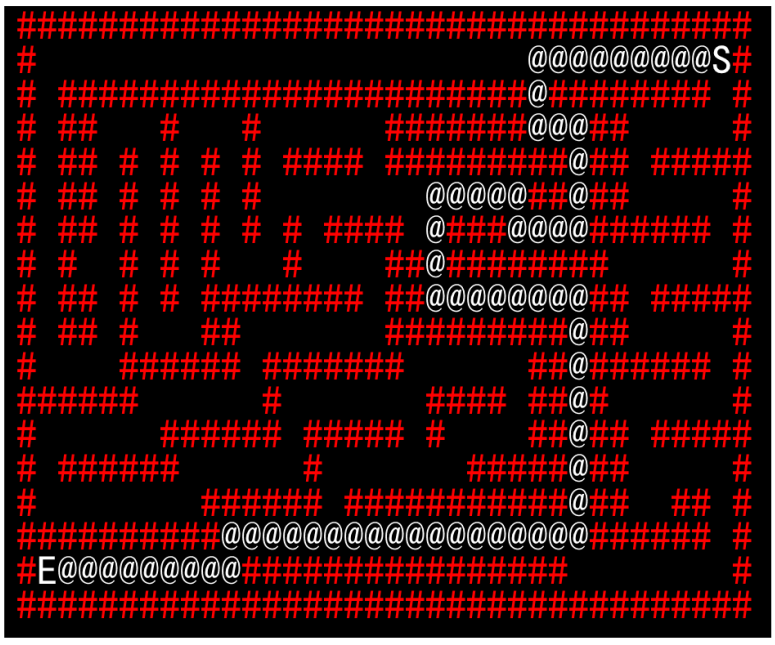
4. 实验结果及分析

4.1.实验结果展示

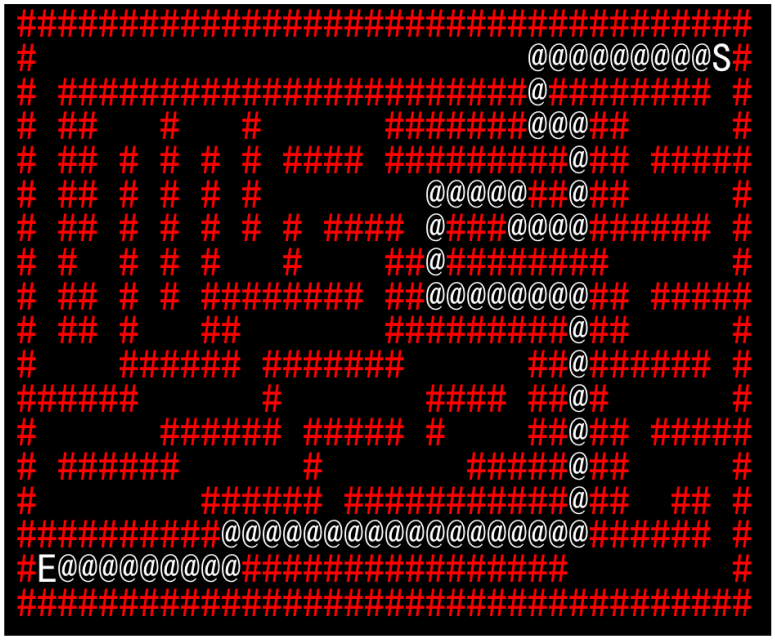
- **深度优先搜索**



- **迭代加深搜索**



- **IDA*搜索**



可以看到，深度优先搜索、迭代加深搜索、IDA*搜索均能找到从起点S到终点E的路径。其中深度优先搜索找到的路径不是最短路径，后两者找到的路径为最短路径。

- **运行时间**

Search	Round	耗时	三轮平均耗时
深度优先搜索	1	0.0170002	0.0183333
	2	0.0160000	
	3	0.0219998	
迭代加深搜索	1	0.5249998	0.4766665
	2	0.4489998	
	3	0.4559998	
IDA*搜索	1	0.0830002	0.0826666
	2	0.0820000	
	3	0.0829997	

可以看到，迭代加深搜索所花时间最长，深度优先搜索所花时间最短。而IDA*搜索所花时间较稳定，多次运行相差不大。

4.2. 评测指标展示

- **深度优先搜索（DFS）**

a) DFS具有完备性。当问题有解时，DFS可以保证找到解。

b) DFS不具有最优性。当问题存在多个解时，DFS不能保证找到最优解。在本次实验中，我们可以看出，DFS找到的路径并不是最短的路径。

c) DFS的时间复杂度为$O(b^m)$，即指数级时间复杂度。其中m指图中最长路径的长度，b指每个节点的子节点的最大数目。

d) DFS的空间复杂度为O(bm)，即线性空间复杂度。

- **宽度度优先搜索（BFS）**

a) BFS具有完备性。当问题有解时，BFS可以保证找到解。

b) BFS具有最优性。当问题存在多个解时，BFS可以保证找到最优解。这是因为BFS按照层次进行遍历，探索出到达目的地的路径就是最短路径。

c) BFS时间复杂度为$O(b^{d+1})$，即指数级时间复杂度。其中b指每个节点的子节点的最大数目，d指初始位置到目标位置最短路径的长度。由于本次实验是在探索节点时判断该节点是否为目的地，所以叶节点都要被探索。

d) BFS的空间复杂度为$O(b^{d+1})$，即指数级空间复杂度。

- **一致代价搜索**

a) 一致代价搜索具有完备性。当问题有解时，一致代价搜索可以保证找到解。

b) 一致代价搜索具有最优性。当问题存在多个解时，一致代价搜索可以保证找到最优解。这是因为每次探索的节点都满足从起点到该节点的距离最短。

c) 一致代价搜索的时间复杂度为$O(b^{d+1})$，与BFS相同。其中b指每个节点的子节点的最大数目，d指初始位置到目标位置最短路径的长度。

d) 一致代价搜索的空间复杂度为$O(b^{c/(s+1)})$。其中c指最短路径的长度，s指顶点到相邻顶点的最短距离。

- **迭代加深搜索**

a) 迭代加深搜索具有完备性。当问题有解时，迭代加深搜索可以保证找到解。

b) 迭代加深搜索具有最优性（在本次实验中，每个节点到相邻节点的距离均相等）。当问题有多个解时，迭代加深搜索可以保证找到最优解。

c) 迭代加深搜索的时间复杂度为$O(b^d)$。其中b指每个节点的子节点的最大数目，d指初始位置到目标位置最短路径的长度。这是因为迭代加深搜索不用扩展深度限制上的节点。

d) 迭代加深搜索的空间复杂度为O(bd)，为指数级空间复杂度。

- **A*搜索**

a) A*搜索具有完备性。当问题有解时，A*搜索可以保证找到解。

b) A*搜索是否具备最优性，取决于h(n)是否是具有可采纳性。在本次实验中，启发式函数采用曼哈顿距离，一定小于或等于实际的最短路径，因此启发式函数是可采纳的，A*搜索具有最优性。当存在多条路径时，A*搜索可以保证找到最短路径。

c) A*搜索的时间复杂度至少为$O(b^{d+1})$。其中b指每个节点的子节点的最大数目，d指初始位置到目标位置最短路径的长度。当h(n)=0时，A*搜索会退化成一致代价搜索。一致代价搜索的时间复杂度的下界也适用于A*搜索。

d) A*搜索的空间复杂度至少为$O(b^{c/(s+1)})$。其中c指最短路径的长度，s指顶点到相邻顶点的最短距离。当h(n)=0时，A*搜索会退化成一致代价搜索。一致代价搜索的空间复杂度的下界也适用于A*搜索。

- **IDA*搜索**

a) IDA*搜索具有完备性。当问题有解时，IDA*搜索可以保证找到解。

b) IDA*搜索具备最优性（在本次实验中，每个节点到相邻节点的距离均相等）。当存在多条路径时，A*搜索可以保证找到最短路径。

c) IDA*搜索的时间复杂度至少为O(bd)。其中b指每个节点的子节点的最大数目，d指初始位置到目标位置最短路径的长度。当h(n)=0时，IDA*搜索会退化成迭代加深搜索。迭代搜索的时间复杂度的下界也适用于IDA*搜索。

d) IDA*搜索的空间复杂度至少为O(bd)。当h(n)=0时，IDA*搜索会退化成迭代加深搜索。迭代搜索的空间复杂度的下界也适用于IDA*搜索。

5. 思考题

这些策略的优缺点是什么？它们分别适用于怎样的场景？

Search	Properties	
深度优先搜索	优点	线性空间复杂度，在一定条件下不必遍历所有分支就可以找到目标节点
	缺点	不具有最优性，得到的路径不一定是最短路径
	适用场景	适用于只需要判断是否能到达目标节点，而不要求得到的路径为最短路径的场景
宽度优先搜索	优点	具有最优性，得到的路径一定是最短路径
	缺点	空间复杂度为指数级别，十分消耗内存
	适用场景	适用于节点的子节点数目不多，图的层次不深
一致代价搜索	优点	可以得到从初始节点到目标节点的最短路径，比宽度优先搜索的空间复杂度低
	缺点	空间复杂度还是指数级别，十分消耗内存 而且时间复杂度比宽度优先搜索高
	适用场景	适用于需要得到最短路径的场景
迭代加深搜索	优点	可以避免陷入无限的分支 可以找到深度最浅的目的节点 线性空间复杂度
	缺点	时间复杂度较高，十分耗时
	适用场景	适用于无限深度的图
A*搜索	优点	减少了不必要的搜索，运行速度快 可以找到成本最低的目的节点
	缺点	需要保存大量的节点信息
	适用场景	适用于需要得到最短路径，同时对搜索时间有限制的场景

IDA*搜索	优点	可以避免陷入无限的分支 可以找到成本最低的目的节点 运行速度快
	缺点	需要保存大量的节点信息
	适用场景	适用于需要得到最短路径，同时对搜索时间有限制的场景，以及无限深度的图