中山大学数据科学与计算机学院

计算机科学与技术 人工智能

本科生实验报告

（2018-2019）学年秋季学期

课程名称：Artificial Intelligence

教学班级	16级计科二班	专业（方向）	计算机科学与技术
学号	16337334	姓名	周启恒

无信息搜索

（1）原理

类型一：BFS

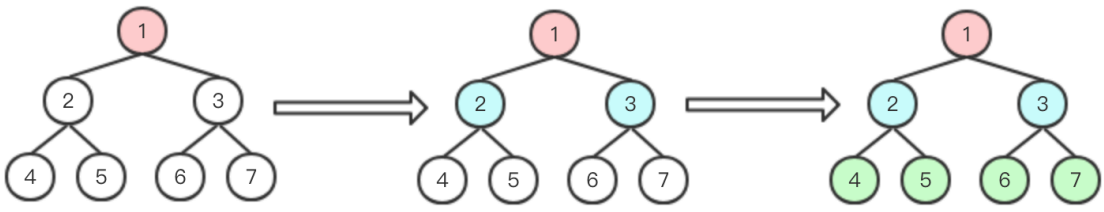
BFS是一种盲目搜索法，按照层结构，系统地展开并检查图中的所有节点。

搜索步骤：

1. 从根节点开始探索，将它的叶节点放入队列中。
2. 从队列中依次拿出节点，对他们分别进行步骤1。

完备性：因为搜寻了整个图/树，一定能找到解。

最优性：宽度优先，每次扩展一层，找到的一定是一条从起点到终点的最短的路径。



类型二：Iterative deepening search

本质：持续执行有深度限制的深度优先搜索，不断增加深度，直至找到目标。

在单次搜索中，按照深度优先搜索的方式，但在整个迭代加深的过程中不断扩展深度，类似宽度优先。这样保证了时间复杂性相同的情况下，减少了空间复杂度。

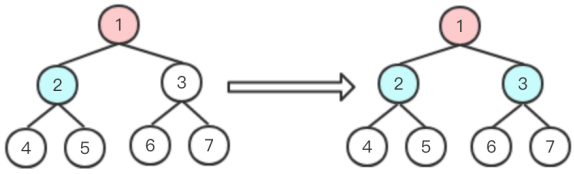
搜索步骤：

1. 设置depth，从1到+∞
2. 按照depth进行深度优先搜索，限制搜索的深度
3. 若找到目标，则停止搜索，返回深度

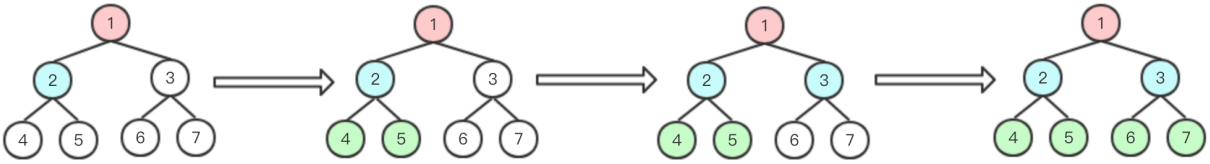
完备性：搜寻了整个图/树，一定能找到解。

最优性：全局上类似宽度优先，每次增加深度，找到的一定是一条从起点到终点的最短的路径。

- depth = 2



- depth = 3



（2）关键代码（带注释）

BFS

```
def BFS(start, end, maze, visit):
    '''
    start:起点
    end:终点
    maze:地图
    visit:标记访问过的节点的矩阵
    '''
    q = Queue()
    q.put(start)

    parent_dict = {}
    path = np.zeros( (len(maze), len(maze[0])) )
    while not q.empty():
        head = q.get()
        #找到终点
        if head == end:
            print("exit found")
            BFS_print_path(parent_dict, path, start, end)
            return
        #向四个邻居扩展,若邻居可扩展，即加入队列中
```

```
        if illegal(head[0] - 1, head[1], maze, visit):
            q.put( (head[0] - 1, head[1]) )
            parent_dict[(head[0] - 1, head[1])] = head
            visit[head[0] - 1][head[1]] = 1

        if illegal(head[0] + 1, head[1], maze, visit):
            q.put( (head[0] + 1, head[1]) )
            parent_dict[(head[0] + 1, head[1])] = head
            visit[head[0] + 1][head[1]] = 1

        if illegal(head[0], head[1] - 1, maze, visit):
            q.put( (head[0], head[1] - 1) )
            parent_dict[(head[0], head[1] - 1)] = head
            visit[head[0]][head[1] - 1] = 1

        if illegal(head[0], head[1] + 1, maze, visit):
            q.put( (head[0], head[1] + 1) )
            parent_dict[(head[0], head[1] + 1)] = head
            visit[head[0]][head[1] + 1] = 1
```

iterative deepening search

```
def iterative_deepening_search(x, y, end, depth, maze, visit):
#指定depth，限制深度优先搜索的深度。
    if (x, y) == end:
        print('exit found')
        print_path(visit)
        return True
    else:
        if depth < 0:
            return False
        #向四个邻居扩展
        if illegal(x + 1, y, maze, visit):
            #标记扩展的邻居
            visit[x+1][y] = 1
            if iterative_deepening_search(x+1, y, end, depth-1, maze,
visit):
                return True
            #回溯
            visit[x+1][y] = 0
        if illegal(x - 1, y, maze, visit):
            visit[x-1][y] = 1
            if iterative_deepening_search(x-1, y, end, depth-1, maze,
visit):
                return True
            visit[x-1][y] = 0
        if illegal(x, y + 1, maze, visit):
            visit[x][y + 1] = 1
```

```
            if iterative_deepening_search(x, y + 1, end, depth-1, maze,
visit):
                return True
            visit[x][y + 1] = 0
        if illegal(x, y - 1, maze, visit):
            visit[x][y - 1] = 1
            if iterative_deepening_search(x, y - 1, end, depth-1, maze,
visit):
                return True
            visit[x][y - 1] = 0
        return False
```

判断节点是否可以扩展

```
def illegal(x, y, maze, visit):
    '''
    x,y表示当前节点的坐标
    '''
    #判断节点是否在迷宫范围内，且可以访问（即迷宫中为0且未访问过）
    if x < 0 or x >= len(maze) or y < 0 or y >= len(maze[0]):
        return False
    elif maze[x][y] != 1 and visit[x][y] == 0:
        return True
    return False
```

调用迭代加深搜索

```
'''
从1开始遍历深度，直至找到合适的深度。
'''
depth = 1
while not iterative_deepening_search(start[0], start[1], end, depth, maze,
visit):
    depth += 1
iterative_deepening_search(start[0], start[1], end, depth, maze, visit)
```

实验结果展示

算法比较：

- 均具有正确性、易读性、健壮性。
- 时间、空间
    - 时间比较：

algorithm	BFS	iterative deepening search
time	$O(b^d)$ / 0.0030769999999999964 s	$O(b^d)$ / 0.054437999999999986 s

- 空间复杂性：

algorithm	BFS	iterative deepening search
space	$O(b^d)$	$O(bd)$

迷宫结果：

```
00000000000000000000000000000000000000
00000000000000000000000000111111111110
00000000000000000000000000100000000000
00000000000000000000000000111000000000
00000000000000000000000000001000000000
00000000000000000000111110011000000000
00000000000000000000100011110000000000
00000000000000000000100000000000000000
00000000000000000000111111110000000000
00000000000000000000000000001000000000
00000000000000000000000000001000000000
00000000000000000000000000001000000000
00000000000000000000000000001000000000
00000000000000000000000000001000000000
00000000000000000000000000001000000000
00000000001111111111111111110000000000
01111111111000000000000000000000000000
00000000000000000000000000000000000000
```

思考题：

6种无信息搜索方法各有什么优缺点？有哪些适用场景？

BFS

- **优点：**
  - 宽度优先搜索按照层序遍历整个图，能找到解，具有完备性。
  - 在路径权重相同的情况下一定能找到最优解，具有最优性。
- **缺点：**在搜索过程中，考虑最坏的情况，需要存储最深的一整层的节点，空间复杂度为$O(b^d)$，其中b为分支数，d为路径的长度。
- **适用场景：**
  - 搜索量较小的场景下，路径权值相同，使用BFS找到最优解。

DFS

- **优点：**每次探索一条路径，考虑最坏情况，需要存储的是最长的一条路径的所有节点，空间复杂度为$O(bd)$.
- **缺点：**在权值相同的情况下，不一定能找到最优解，即不满足最优性。

- **适用场景：**相对BFS，它的空间复杂度要求较低，可以适用于搜索量更大的场景。

一致代价

- **优点：**
  - 与BFS相同，可以找到解，具有完备性。
  - 按照权值进行选择，能找到最优解，具有最优性。
- **缺点：**与BFS相同，在搜索过程中，考虑最坏的情况，需要存储最深的一整层的节点，空间复杂度为$O(b^d)$，其中b为分支数，d为路径的长度。
- **适用场景：**
  - 与BFS相同，适用于小规模的搜索，且在路径权值不同的时候也可以找到最优解。

迭代加深

- **优点：**
  - 每次搜索采用深度优先搜索，控制了空间复杂度。
  - 而在全局上限制深度，相当于宽度优先搜索，可以找到最优解。同时具备完备性与最优性。
- **缺点：**在搜索量规模非常大的情况下，迭代次数可能非常大，耗时长。
- **适用场景：**
  - 在搜索量较大的情况，可代替BFS，保证了完备性和最优性，也降低了空间复杂度。

双向搜索

- **优点：**
  - 双向搜索，从起点和终点同时进行BFS搜索扩展，直到相遇，具有最优性和完备性。
  - 空间复杂度从BFS的$O(b^d) \rightarrow O(b^{d/2})$
- **缺点：**空间复杂度的优化不到位，还是需要指数级的存储空间，不适用于规模较大的场景。
- **适用场景：**在BFS的适用场景下，换为双向搜索，降低一部分空间复杂度。