EDA 软件设计 I

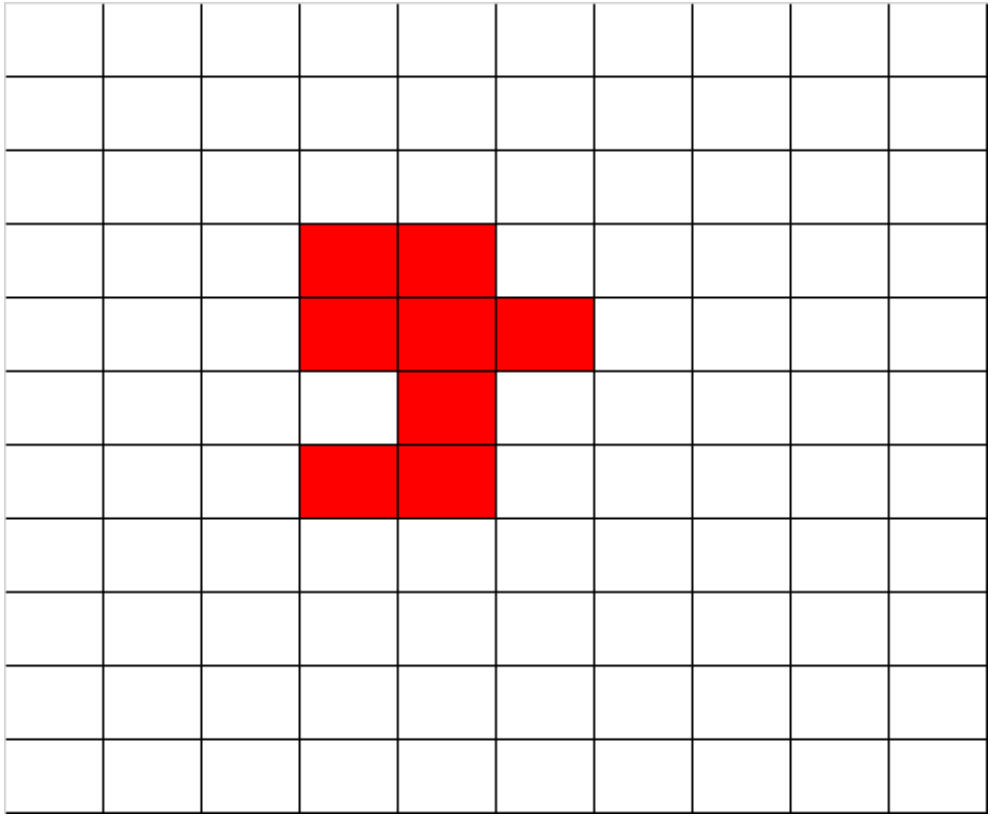
Lecture 7

Review：算法应用

- 实际（工程）问题不会提示你运用什么算法或者采用哪个算法的策略
- 需要你对算法融会贯通、举一反三，来解决实际问题

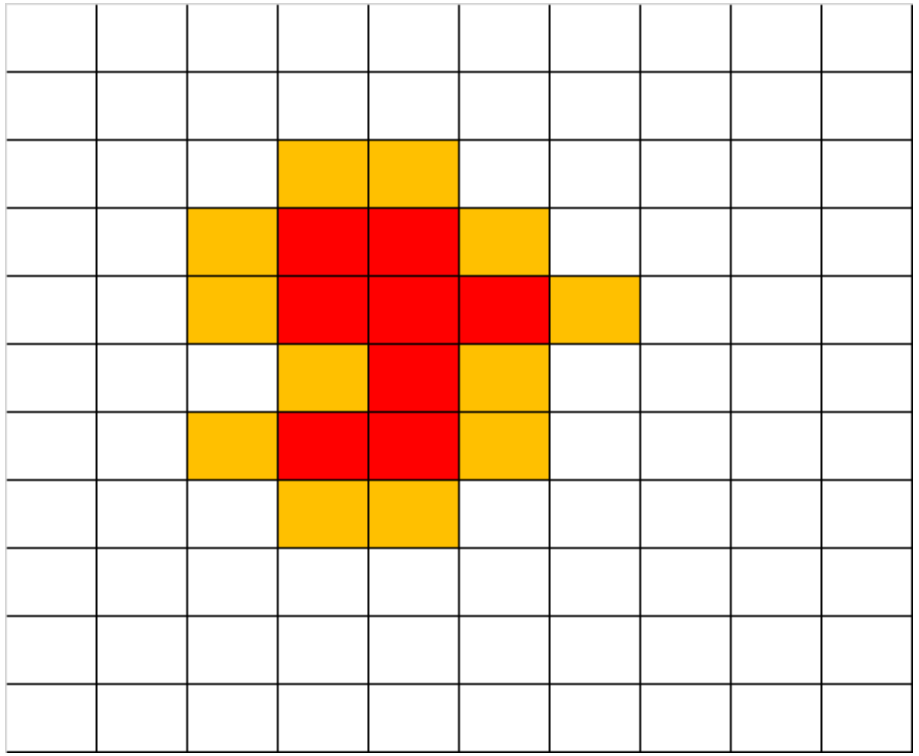
1. 熟悉一个算法**最典型**的应用场景
   - 例如BFS: 最短路径问题（无权图）、连通性检测
2. 建立系统的算法问题解决框架，从**问题理解、模型建立、算法设计、实现优化**到**结果分析**

Review: 扩展练习



左边的矩阵内存在一个红色区域，求出红色区域距离矩阵边缘的最短距离

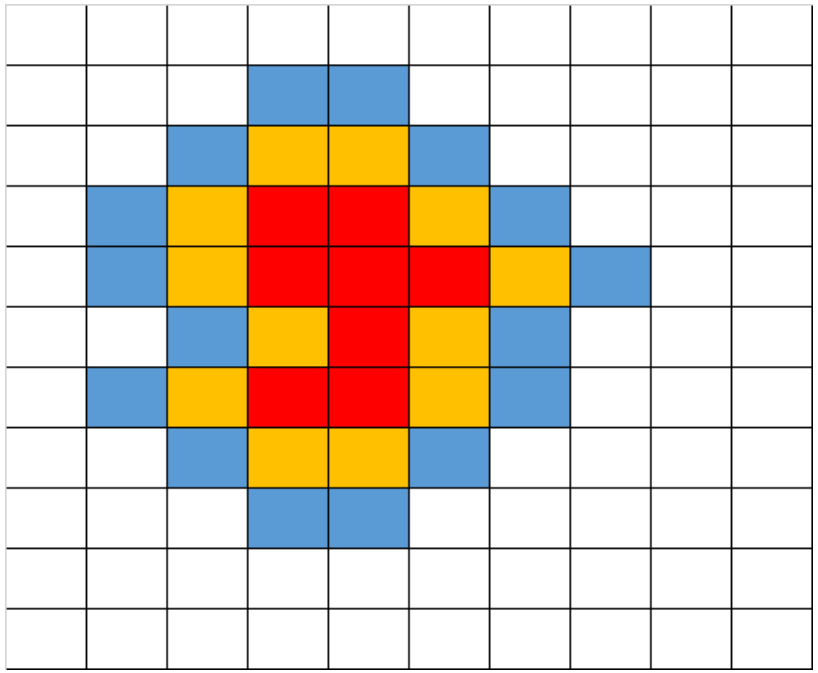
Review：扩展练习

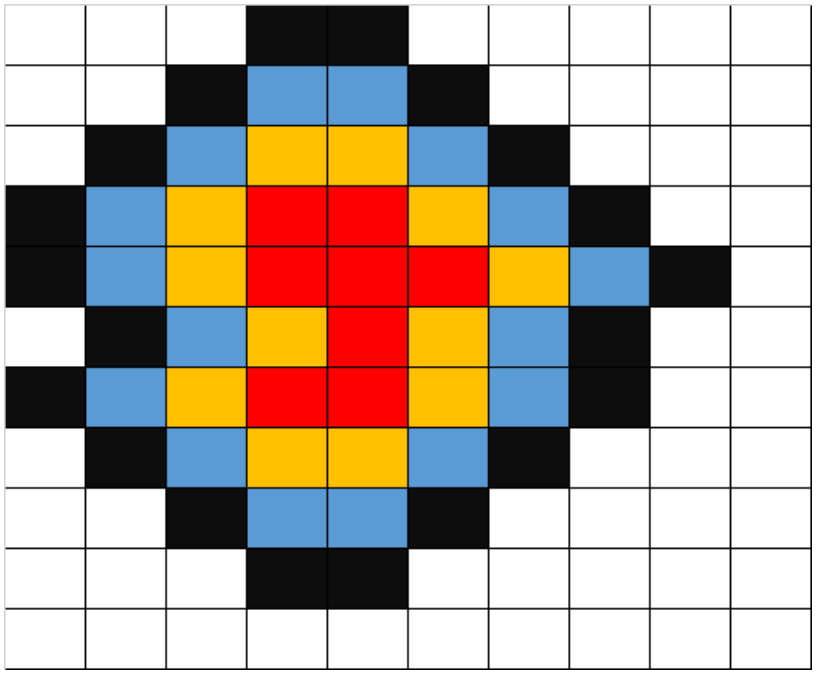


黄色区域：bfs迈出的第一步

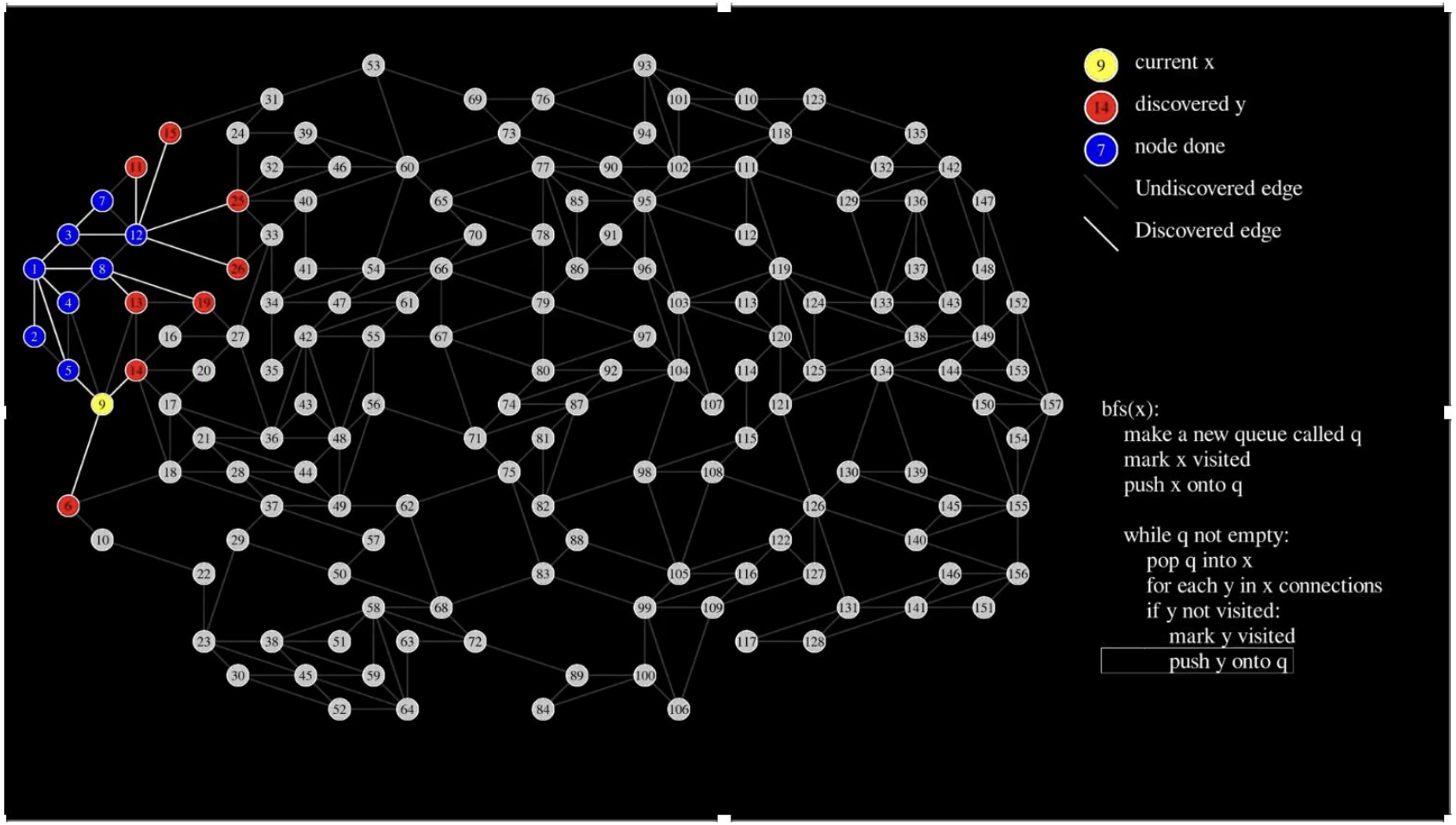
如果红色区域被定义为第一层，那么黄色区域就是bfs的第二层

Review: 扩展练习



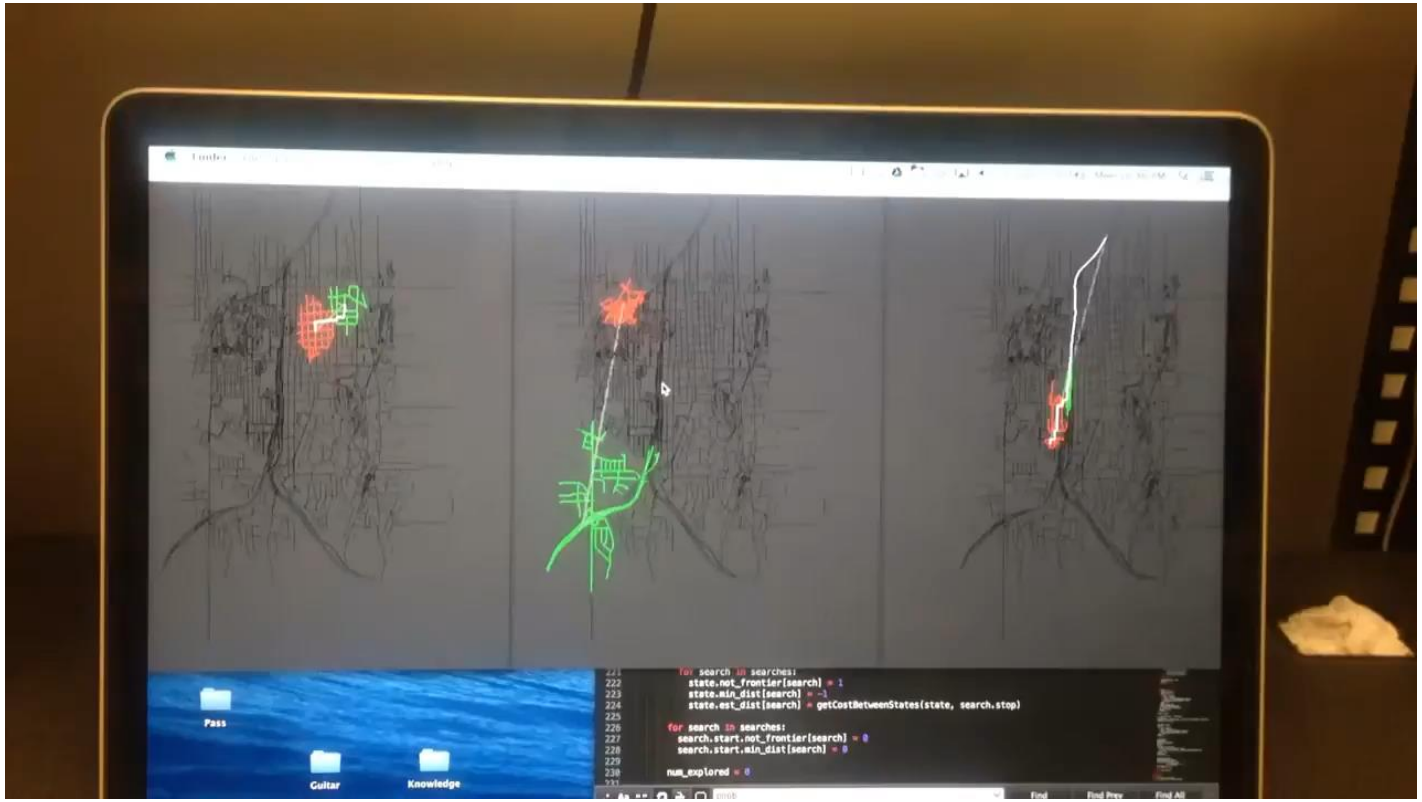


Review: BFS扩展



Review: 双向 BFS

- 同时从起点和终点进行搜索，缩短搜索路径，提高搜索效率
- 适用于在大规模图中寻找最短路径的问题



Review: 算法四大核心要素

Review: 算法核心要素

原理：概念和策略

实现：将算法原理转化为具体代码或程序的过程

分析：评估算法在时间和空间上的效率，以衡量其性能和可行性

应用：在实际场景中运用算法来解决各种具体问题的过程

BFS Done

深度优先搜索

深度优先搜索

- Depth-First Search（DFS）
- 与 BFS 对比：
  - BFS 像一个扫街的人，他会先把离自己最近的区域全部探查完，然后再向外扩展。
  - DFS 则类似于一个探险家，走一条路径走到黑，深入到底后才回头，继续走另一条路径。

算法核心四要素 @ DFS

Review：算法原理（概念和策略）

- 算法原理：解决问题的核心思路的抽象描述，强调解决问题的思维方式和逻辑框架
  - ① 自然语言描述
  - ② 图示、可视化描述
  - ③ 伪代码简化版描述
- 常见的算法设计策略：
  1. 分治法（divide and conquer）
  2. 贪心（Greedy）
  3. 动态规划（dynamic programming）
  4. 回溯（backtracking）
- 锻炼抽象思维：从具体问题中抽象出通用的算法模型

算法原理 @ DFS（同学们版）

同学 A

- “DFS 将节点分组标记或者对边分组标记”

DFS 本身的主要过程是深入探索节点，而不是对节点或边进行分组标记。

同学 B

- “DFS 优先走通一条路”

DFS 确实会沿着一条路径尽可能深入，直到无法继续，再回溯到上一个节点

同学 C

- “DFS 优先查找层数深”

反映了 DFS 的深度优先特性，相比于广度优先搜索（BFS）逐层遍历，DFS 更注重深入到更深的层级

同学 D

- “DFS一条线贯穿到尾”

描述不准确，但描述了 DFS 一直深入的特点到底

同学 E

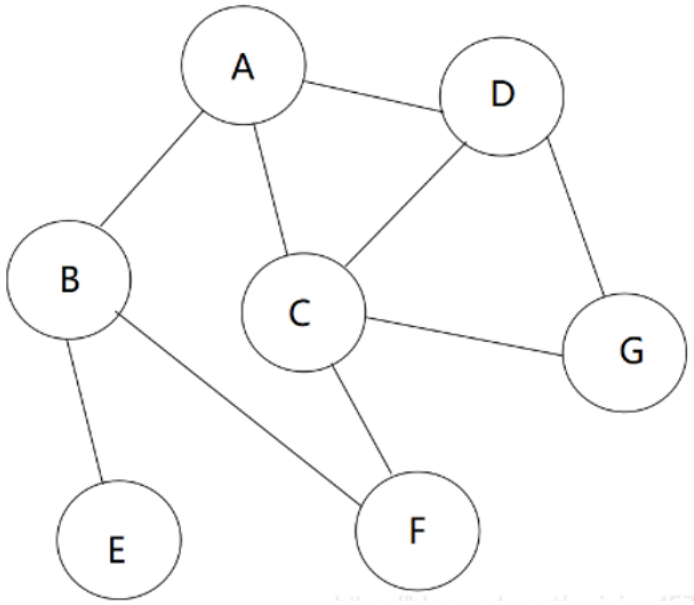
- “DFS一直往后搜索”

反应了DFS 持续深入的过程，但“往后”一词是什么意思？在图结构上有「后」这个概念吗？

算法原理 @ DFS

- **自然语言描述**——short：从起始节点出发，沿着一条路径不断深入到树或图的“最深处”，直到不能再深入为止; 然后回溯到前一个节点，继续探索其他未访问的路径。
- **自然语言描述**——in detail:
  1. DFS 从某个起始节点开始，选择一个可到达的邻居节点（如果有多个节点可选，任选一个）并继续深入
  2. 沿着当前路径向前走，直到到达某个节点，这个节点没有任何未访问的邻居节点
  3. 返回上一个节点，检查是否有其他未访问的邻居节点。如果有，就沿着新的路径继续深入；如果没有，继续回溯，直到返回到最开始的节点
  4. 重复上述过程：直到所有节点都被访问过

算法思想 @ DFS



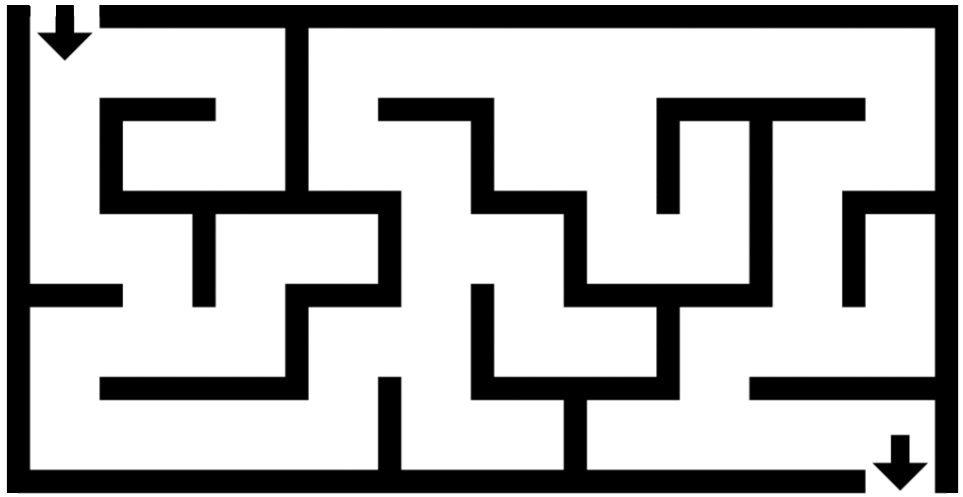
DFS遍历在**small graph**上的展示，

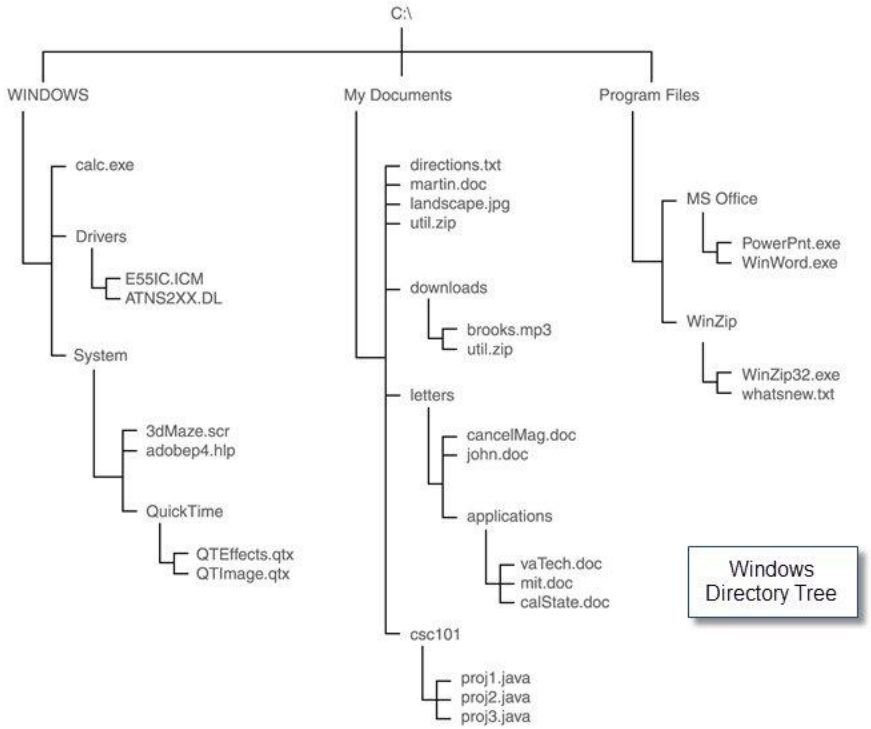
假设起始点是**G**：

1. G → D → A → B → E → F → C ✔
2. G → D → A → B → E → C → F ✘
3. G → C → A → D → F → B → E ✘
4. G → C → A → D → B → E → F ✔

算法原理 @ DFS

- **类比：迷宫找出口、目录文件夹查找**



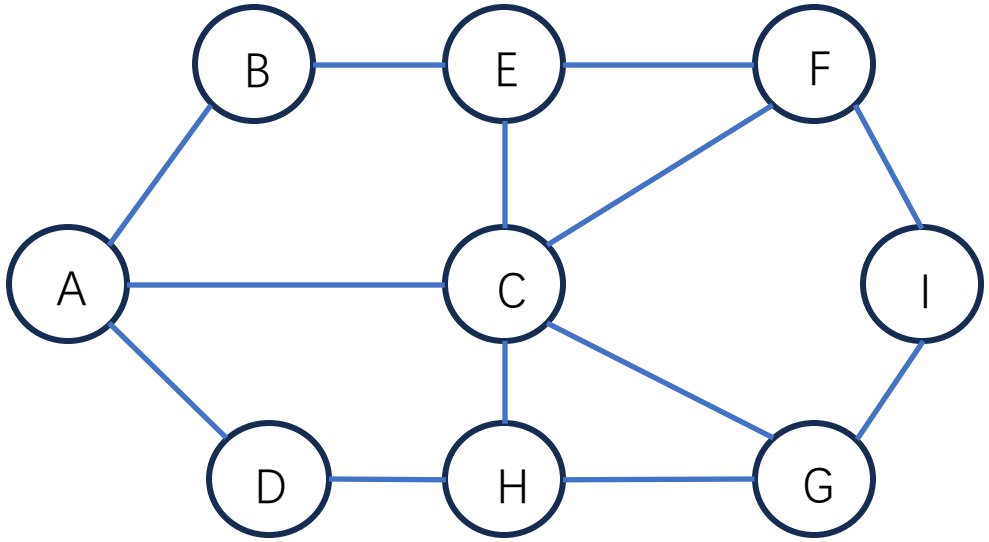


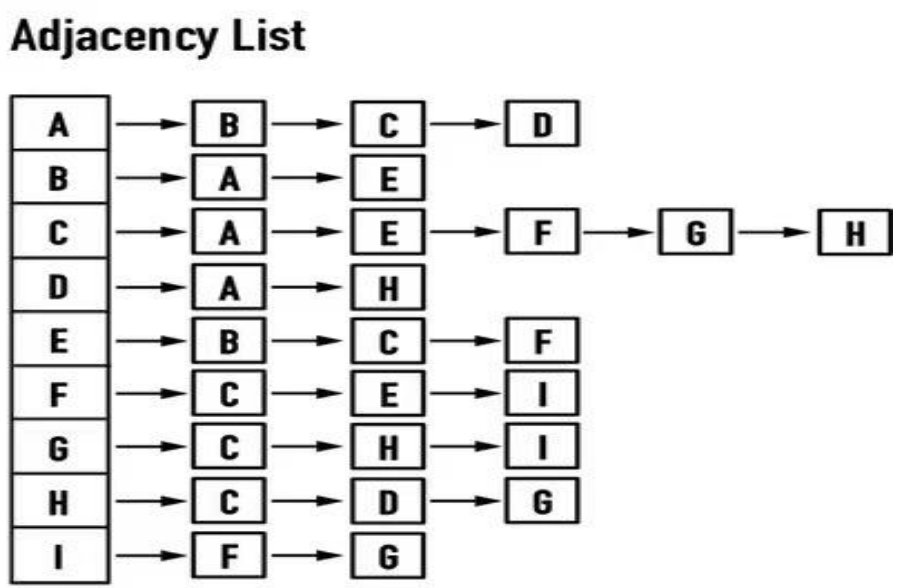
算法核心四要素 @ DFS

算法实现 @ DFS遍历

- DFS 遍历:
  - input: Graph + 起始节点
  - output: 用DFS的方式获得的图节点访问顺序
  - 算法名称：**DFS traversal （深度优先遍历）**
- DFS 遍历实现（经典实现）要点:
  - 可以用**栈(Stack: FILO)**来实现
  - 维护一个**“已访问”列表**
  - 也可以用**递归**实现

DFS遍历算法实现可视化（by Stack）





Input: 邻接表+起始节点A

DFS遍历算法实现可视化（by Stack）

S：用于实现算法的数据结构stack

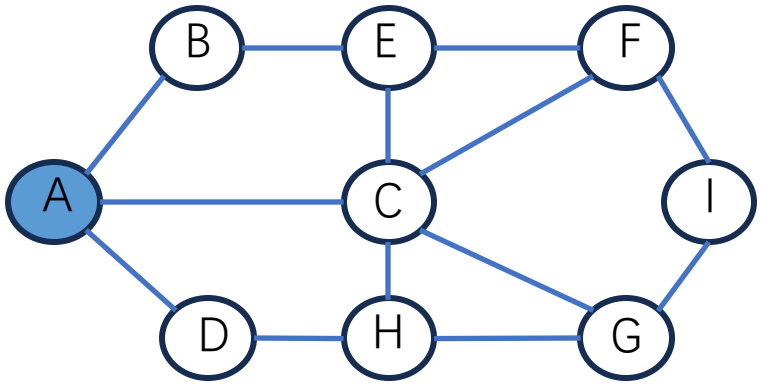
Visited：维护的“已访问”列表（数据结构：集合）

Res：输出结果，可用列表（Python）

伪代码：

```
当栈 S 非空时:
    从栈 S 中pop出最后进入的节点作为当前节点
    对于当前节点的每个邻居:
        如果邻居不在 Visited 集合中:
            将邻居节点推入栈 S
            将邻居节点加入 Visited 集合
    将当前节点加入 Res 列表
```

可视化过程展示：体会 stack 是如何起到作用来实现 DFS的算法原理的




```
当栈 S 非空时:
    从栈 S 中pop出最后进入的节点作为当前节点
    对于当前节点的每个邻居:
        如果邻居不在 Visited 集合中:
            将邻居节点推入栈 S
            将邻居节点加入 Visited 集合
    将当前节点加入 Res 列表
```

初始状态

Visited = 【A】

S = A

Res = 空

当前节点：A
在邻接表里读出A的
邻居: B,C,D
其中B,C,D不在
Visited中

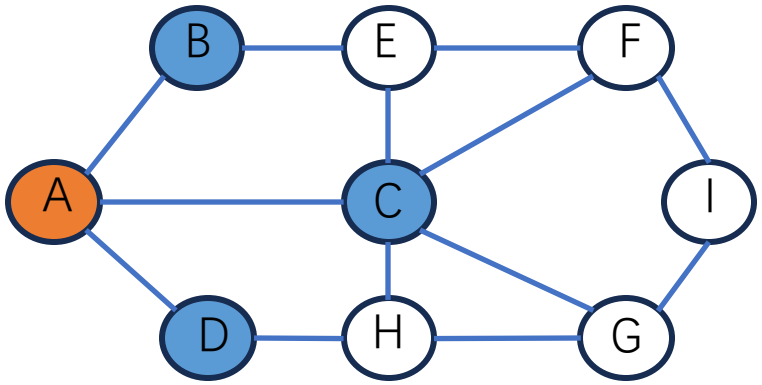
下一状态

Visited = 【A,B,C,D】

D
C
B

S =

Res = A




```
当栈 S 非空时:
    从栈 S 中pop出最后进入的节点作为当前节点
    对于当前节点的每个邻居:
        如果邻居不在 Visited 集合中:
            将邻居节点推入栈 S
            将邻居节点加入 Visited 集合
    将当前节点加入 Res 列表
```

当前状态

Visited = 【A,B,C,D】

S =

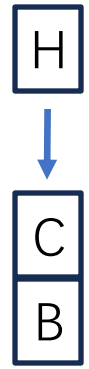
Res =

当前节点：D
在邻接表里读出D的邻居: A,H
其中H不在**Visited**中

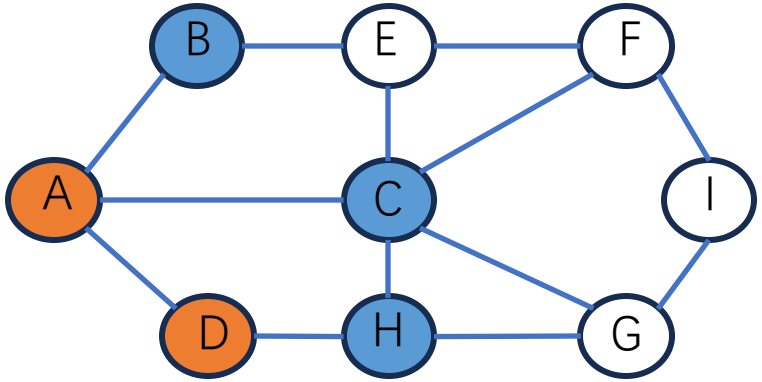
下一状态

Visited = 【A,B,C,D,H】

S =



Res =




```
当栈 S 非空时:
    从栈 S 中pop出最后进入的节点作为当前节点
    对于当前节点的每个邻居:
        如果邻居不在 Visited 集合中:
            将邻居节点推入栈 S
            将邻居节点加入 Visited 集合
    将当前节点加入 Res 列表
```

当前状态

Visited = 【A,B,C,D,H】

S =

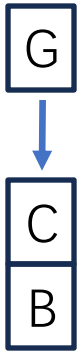
Res =

当前节点：H
在邻接表里读出H的邻居: C,D,G
其中G不在**Visited**中

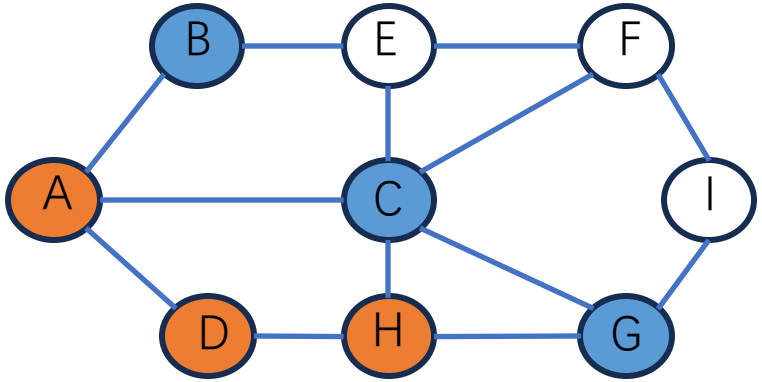
下一状态

Visited = 【A,B,C,D,H,G】

S =



Res =




```
当栈 S 非空时:
    从栈 S 中pop出最后进入的节点作为当前节点
    对于当前节点的每个邻居:
        如果邻居不在 Visited 集合中:
            将邻居节点推入栈 S
            将邻居节点加入 Visited 集合
    将当前节点加入 Res 列表
```

当前状态

Visited = 【A,B,C,D,H,G】

S =

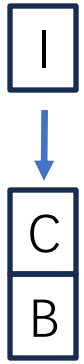
Res =

当前节点：G
在邻接表里读出G的
邻居: C,H,I
其中I不在**Visited**中

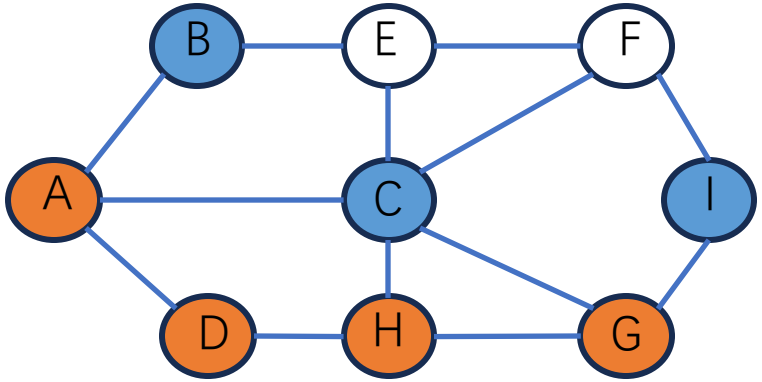
下一状态

Visited =
【A,B,C,D,H,G,I】

S =



Res =




```
当栈 S 非空时:
    从栈 S 中pop出最后进入的节点作为当前节点
    对于当前节点的每个邻居:
        如果邻居不在 Visited 集合中:
            将邻居节点推入栈 S
            将邻居节点加入 Visited 集合
    将当前节点加入 Res 列表
```

当前状态

Visited =
【A,B,C,D,H,G,I】

S =

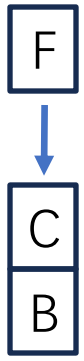
Res =

当前节点：I
在邻接表里读出 I 的邻居: F,G
其中 F 不在**Visited**中

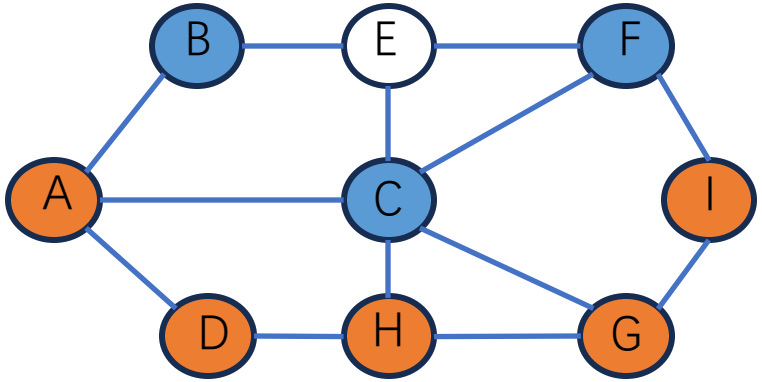
下一状态

Visited =
【A,B,C,D,H,G,I,F】

S =



Res =




```
当栈 S 非空时:
    从栈 S 中pop出最后进入的节点作为当前节点
    对于当前节点的每个邻居:
        如果邻居不在 Visited 集合中:
            将邻居节点推入栈 S
            将邻居节点加入 Visited 集合
    将当前节点加入 Res 列表
```

当前状态

Visited =
【A,B,C,D,H,G,I,F】

S =

Res =

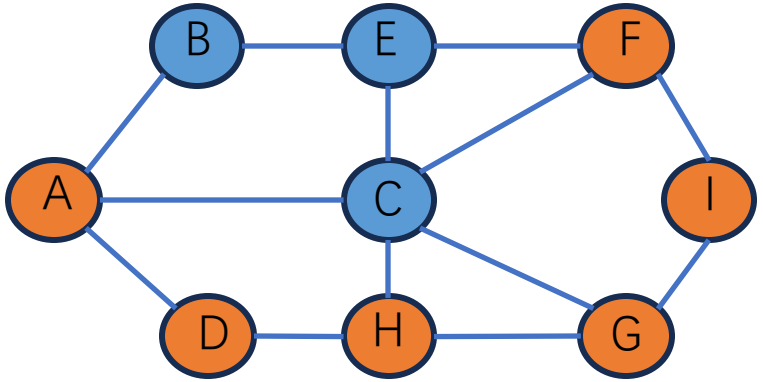
当前节点：F
在邻接表里读出 F 的邻居: C,E,I
其中 E 不在**Visited**中

下一状态

Visited =
【A,B,C,D,H,G,I,F,E】

S =

Res =




```
当栈 S 非空时:
    从栈 S 中pop出最后进入的节点作为当前节点
    对于当前节点的每个邻居:
        如果邻居不在 Visited 集合中:
            将邻居节点推入栈 S
            将邻居节点加入 Visited 集合
    将当前节点加入 Res 列表
```

当前状态

Visited =
【A,B,C,D,H,G,I,F,E】

S =

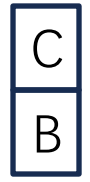
Res =

当前节点：E
在邻接表里读出E 的邻居: B,C,F
没有不在**Visited**中的邻居

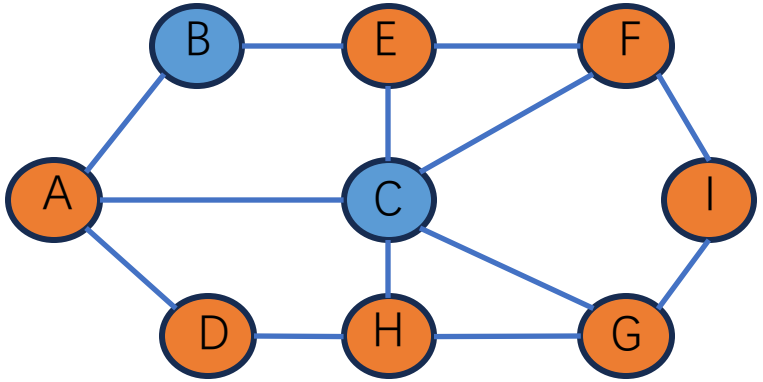
下一状态

Visited =
【A,B,C,D,H,G,I,F,E】

S =



Res =




```
当栈 S 非空时:
    从栈 S 中pop出最后进入的节点作为当前节点
    对于当前节点的每个邻居:
        如果邻居不在 Visited 集合中:
            将邻居节点推入栈 S
            将邻居节点加入 Visited 集合
    将当前节点加入 Res 列表
```

当前状态

Visited =
【A,B,C,D,H,G,I,F,E】

S = C, B

Res = A D H G I F E

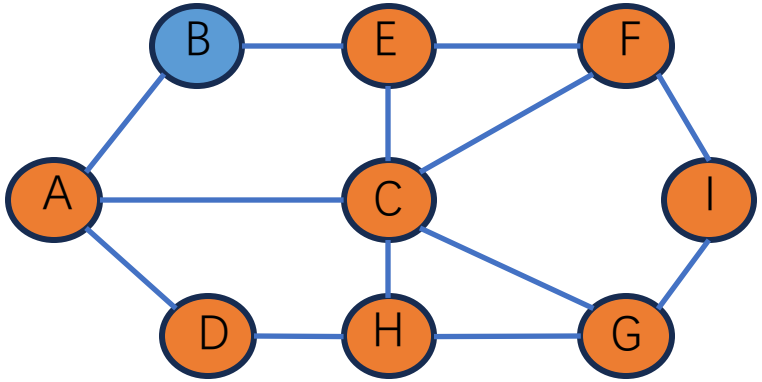
当前节点：C
在邻接表里读出 C 的邻居:A,E,H,F,G
没有不在**Visited**中的邻居

下一状态

Visited =
【A,B,C,D,H,G,I,F,E】

S = B

Res = A D H G I F E C




```
当栈 S 非空时:
    从栈 S 中pop出最后进入的节点作为当前节点
    对于当前节点的每个邻居:
        如果邻居不在 Visited 集合中:
            将邻居节点推入栈 S
            将邻居节点加入 Visited 集合
    将当前节点加入 Res 列表
```

当前状态

Visited =
【A,B,C,D,H,G,I,F,E】

S = B

Res = A D H G I F E C

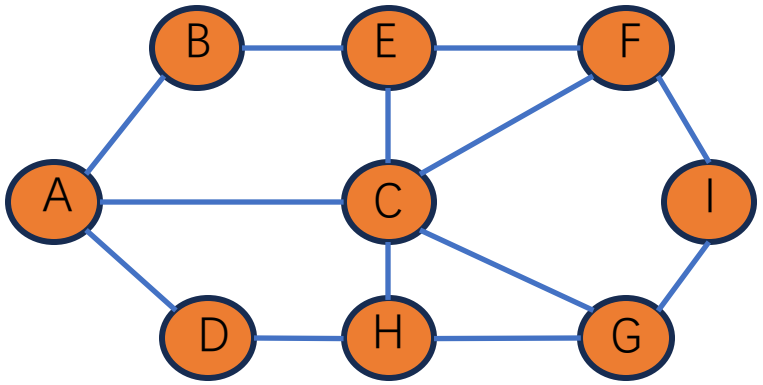
当前节点：B
在邻接表里读出 B 的邻居:A,E
没有不在**Visited**中的邻居

下一状态

Visited =
【A,B,C,D,H,G,I,F,E】

S = 空

Res = A D H G I F E C B




```
当栈 S 非空时:
    从栈 S 中pop出最后进入的节点作为当前节点
    对于当前节点的每个邻居:
        如果邻居不在 Visited 集合中:
            将邻居节点推入栈 S
            将邻居节点加入 Visited 集合
    将当前节点加入 Res 列表
```

当前状态

Visited =
【A,B,C,D,H,G,I,F,E】

S = 空

Res = | A | D | H | G | I | F | E | C | B |

此刻S为空，跳出循环，返回 Res

输出结果为遍历顺序：

| A | D | H | G | I | F | E | C |

实现细节

维护已访问节点，有别的方式吗？