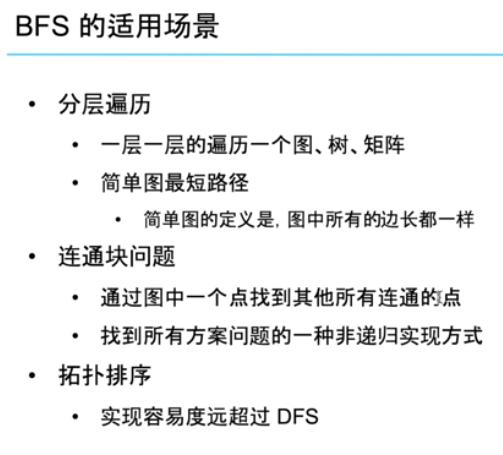
宽度优先搜索（BFS）：适用于分层遍历、连通块问题、拓扑排序。



lintcode 69：二叉树的层序遍历使用BFS，有三种方法：

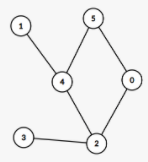
1. 单队列（推荐）
2. 双队列
3. DummyNode（哨兵结点）

DummyNode主要是在链表中使用，DummyNode.Next永远指向链表的头部，而DummyNode本身不变，不会被删除掉，因为其不是链表的一部分。

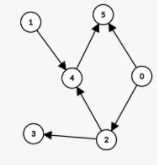
图(Graph)的基本知识：图在离线数据中的表示方法为。E表示Edge，表示边；V是Vertex，表示顶点，图其实是边和顶点的集合。

在图中，每一条边都是一个点对，其中 。例如下面的无向图中，(1, 4)就表示顶点1和顶点4之间的边。图中的顶点有邻接的概念，对于两个顶点v和w，如果，即v和w构成的点对如果属于边的集合，则v和w就是邻接点，下图中(1, 4)和邻接点，(4, 5)也为邻接点，但(1, 5)不是邻接点，它们构成的点对不是一条边。

图分为：无向图和有向图。



无向图



有向图

从上述邻接点的概念中看出，在无向图中，如果v和w构成的是一条边，则v和w是互为邻接点的，但在有向图中，则未必如此，因为边带有方向，所以顶点v的邻结点是w，但反过来则不一定成立。

BFS在两种图上都适用。

树其实是一种特殊的图，在二叉树中进行BFS，不需要考虑已访问过的结点，因为不存在一个结点的子结点的子结点是它自己的情况，但是在图的BFS中，一个结点的邻居结点可以是它自己（即结点的一条边指向它自己，构成自环），所以在图的BFS中需要使用哈希map来记录已访问的结点，C++中为unordered\_map，python中为dict。

图的顶点可以没有边，所以图的顶点减一等于边的数量时，这个图也不一定是一棵树，例如：

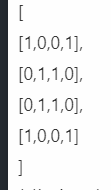


上述的图中，有4个结点，3条边，其中顶点4是没有边的，但它不是一颗树，它是一个不连通的图。

图的数据结构主要由两种：

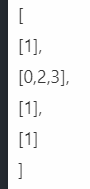
1. 邻接矩阵（空间耗费比较大）
2. 邻接表（常用）

邻接矩阵一般是一个二维数组，如下图所示的邻接矩阵A



A[u][v]=1，表示(u,v)是一条边，A[u][v]=0，表示(u,v)不是一条边，每个点默认和自己是一条边，上述的图中，A[0][3] = 1，顶点0和顶点3是连通的，之间有一条边，A[1][2] = 1，顶点1和顶点2是连通的，之间有一条边。从上述矩阵看出，这应该是一个无相同，顶点0和顶点3互为邻接点，顶点1和顶点2互为邻接点，无向图的邻接矩阵是一个对称矩阵。

邻接矩阵耗费空间为，尤其是稀疏图上，浪费的空间比较严重，而邻接表则比较常用。



邻接表中每个点存储自己有哪些邻接点，第一行的[1]表示顶点0和顶点1邻接，第二行的[0,2,3]表示顶点1和顶点0，顶点1和顶点2，顶点1和顶点3都邻接。

BFS使用场景：

1. 连通块问题：（并查集）
   1. 通过一个点找到图中连通的所有点
   2. 非递归的方式找所有方案
2. 分层遍历：
   1. 图的层次遍历
   2. 简单图最短路径
3. 拓扑排序：依赖性问题
   1. 求任意拓扑排序
   2. 求是否有拓扑序
   3. 求字典序最小的拓扑序
   4. 求是否唯一拓扑序

BFS无法解决最长路径。

图可以分层，用动态规划。

图的分层：路径有一定方向性，不能绕圈。

图的BFS中，一般会用到哈希表来存储访问过的结点。

最简洁的BFS模板：

## 宽度优先搜索-简单的模板：最短路径问题。

## step 1. 初始化，把初始结点放入到deque，如果有多个，就都放进去

## 在distance标记初始结点的距离为0，distance的作用有两个

## 一是标记结点已访问过，二是记录结点距离

queue = collections.deque([node])

distance = {node : 0}

## step 2. 不断访问队列，添加结点到队列中并pop出之前的结点

while queue:

node = queue.popleft()

## step 3. 拓展相邻的结点，将pop出的结点的相邻结点加入到队列中

for neighbor in node.get\_neighbors():

if neighbor in distance: ## 该邻居已访问过

continue

distance[neighbor] = distance[node] + 1

queue.append(neighbor)

在Python中，node一般是一个元组对象，可以作为dict的key。

node是类的实例对象时，也可以作为dict的key。

宽度优先搜索的时间复杂度：

在图上进行BFS时，时间复杂度为：。其中n为点数，m为边数。最坏的情况为，主要是密集图。

分析如下：在上面代码中，，虽然有两层循环，外层的while和内层的for，这是因为外层的while循环中弹出结点的pop操作会执行n次， n个结点都会只入队一次，这是因为有哈希表distance保证的，而最内层的循环体中，在distance的作用下，每条边的两个结点会分别访问这条边一次，也就是说，内层的for循环访问的neighbors结点总共会有2\*m个(两个结点互为邻接点)， 这样时间复杂度就为。最坏情况下是，每个结点都有n个邻接点，m就是n\*n，时间复杂度就为。

拓扑排序：能用BFS就用BFS，实在不行用DFS，非递归的DFS，太难以理解，面试官也未必能明白，而递归的DFS，比较容易stackoverflow。

基本的定义：

入度(In-degree)：在有向图中，如果图中的1个点有一条边指向它，则其入度+1，假如没有任何一个边指向它，入度为0.



有向图：图中的边是有方向的，即点A指向点B，A的邻接点是B，反过来则未必成立。

拓扑排序并非传统的排序，而是图中所有顶点的线性序列，这个序列满足以下两个条件：

1. 每个顶点出现且只出现1次
2. 若存在一条从顶点A到顶点B的边，在序列中点A在点B的前面

所以拓扑排序只针对有向无环图。如果有环，肯定不满足条件2.



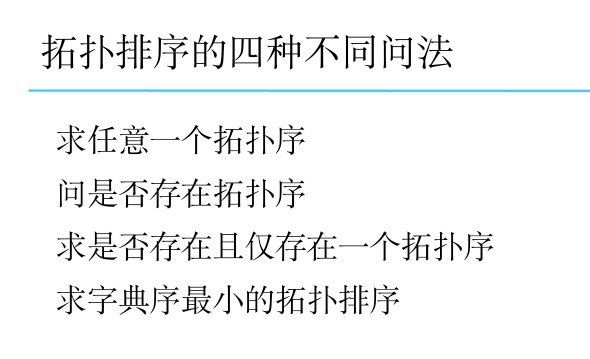
有环图

算法描述：

1. 统计每个点的入度
2. 将每个入度为0的点放入队列中作为起始结点
3. 不断从队列中拿出一个点，去掉这个点的所有边（指向其他点的边），其它的点的入度-1
4. 一旦发现新的入度为0的点，丢回队列中。

最后的拓扑序就是进入队列的顺序，或者弹出队列的顺序。

一个图中，拓扑排序有可能存在多个，也有可能不存在。



宽度优先搜索找所有方案：BFS通常找一条路径，一种方案。BFS擅长找连通块的问题，把路径看成点，把路径的变化关系看成点的连接关系，这样找所有路径就成了找连通块的问题。

lintcode 17：例 集合为{1,2,3}

第一种搜索树的画法：N叉树



第二种画法：完全二叉树

