

性价比之王的宽度优先搜索

主讲人 令狐冲
课程版本 v7.0

版权声明

九章的所有课程均受法律保护，不允许录像与传播录像
一经发现，将被追究法律责任和赔偿经济损失

下载 APP

邀请有礼



扫码下载九章算法 APP
学习交流领取专享福利

LintCode 搜题

今日看题 0 分钟

随机一题

我的题单

收藏题解

学习笔记

类型筛选

题目标签

综合排序

1905 · 字符删除

简单 · 55% 通过率 · 3 题解

1904 · 放小球

困难 · 51% 通过率 · 3 题解

1903 · 部门统计

简单 · 45% 通过率 · 6 题解

1902 · 寻找Google

简单 · 43% 通过率 · 3 题解

1901 · 有序数组的平方

简单 · 76% 通过率 · 5 题解

1900 · 基因相似度

困难 · 46% 通过率 · 3 题解

</>

发现

消息

我的

<

137 · 克隆图

...

题目

题解

笔记

脸书

Depth-first Search

优步

谷歌

哈希表

哈希表

Breadth-first Search

哈希表

脸书

谷歌

Pocket Gems

中等 · 37% 通过率

添加题单

描述

克隆一张无向图. 无向图的每个节点包含一个 label 和一个列表 neighbors. 保证每个节点的 label 互不相同.

你的程序需要返回一个经过深度拷贝的新图. 新图和原图具有同样的结构, 并且对新图的任何改动不会对原图造成任何影响.

说明

你需要返回与给定节点具有相同 label 的那个节点.

示例

样例1

输入:

{1,2,4#2,1,4#4,1,2}

输出:

{1,2,4#2,1,4#4,1,2}

解释:

1-----2

输入评论...

...

<

题解详情

...

令狐冲

发布于: 2018-08-25

使用宽度优先搜索 BFS 的版本。

第一步: 找到所有的点

第二步: 复制所有的点, 将映射关系存起来

第三步: 找到所有的边, 复制每一条边

```
class Solution:
    def cloneGraph(self, node):
        root = node
        if node is None:
            return node

        # use bfs algorithm to traverse the graph
        nodes = self.getNodes(node)

        # copy nodes, store the old->new mapping
        mapping = {}
        for node in nodes:
            mapping[node] = UndirectedGraphNode(

        # copy neighbors(edges)
        for node in nodes:
            new_node = mapping[node]
            for neighbor in node.neighbors:
                new_neighbor = mapping[neighbor]
                new_node.neighbors.append(new_ne

        return mapping[root]

    def getNodes(self, node):
        q = collections.deque([node])
        result = set([node])
        while q:
            head = q.popleft()
```

输入评论...

☆

👍

...



问最短路径

除了 **BFS** 还有可能是什么算法？

问最短路径

简单图：BFS

复杂图：Floyd, Dijkstra, Bellman-ford, SPFA

面试中一般不考复杂图最短路径问题

问最长路径

BFS 是否可以做？
应该用什么算法呢？

问最长路径

图可以分层：动态规划 **Dynamic Programming**

图不可以分层：深度优先搜索 **DFS**

分层的意思是：路径有一定方向性，不能绕圈

第 i 层的点只能走到第 $i+1$ 层不能回到底 $i-1$ 层

二叉树 vs 图的 BFS

有什么区别？

哈希表

图中存在环，同一个节点可能重复进入队列

Java: *HashMap / HashSet*

Python: *dict / set*

C++: *unordered_map / unordered_set*



Clone Graph

<http://www.lintcode.com/problem/clone-graph/>

<http://www.jiuzhang.com/solutions/clone-graph/>

连通块问题

返回一个经过深度拷贝的新图. 新图和原图具有同样的结构, 并且对新图的任何改动不会对原图造成任何影响.

这份代码有什么问题

```
def cloneGraph(self, node):
    if not node:
        return None
    queue = [node]
    start = 0
    mapping = {}
    while start < len(queue):
        curt_node = queue[start]
        start += 1
        if curt_node in mapping:
            new_node = mapping[curt_node]
        else:
            new_node = UndirectedGraphNode(curt_node.label)
            mapping[node] = new_node
        for neighbor in curt_node.neighbors:
            if neighbor in mapping:
                new_neighbor = mapping[neighbor]
            else:
                new_neighbor = UndirectedGraphNode(neighbor.label)
                mapping[neighbor] = new_neighbor
                queue.append(neighbor)
            new_node.neighbors.append(new_neighbor)
    return mapping[node]
```

```
public class Solution {
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
        if (node == null) {
            return null;
        }
        Queue<UndirectedGraphNode> queue = new ArrayDeque<>();
        int start = 0;
        queue.offer(node);
        Map<UndirectedGraphNode, UndirectedGraphNode> mapping = new HashMap<>();
        while (!queue.isEmpty()) {
            UndirectedGraphNode curtNode = queue.poll();
            UndirectedGraphNode newNode;
            if (mapping.containsKey(curtNode)) {
                newNode = mapping.get(curtNode);
            }
            else {
                newNode = new UndirectedGraphNode(curtNode.label);
                mapping.put(curtNode, newNode);
            }
            UndirectedGraphNode newNeighbor;
            for (UndirectedGraphNode neighbor : curtNode.neighbors) {
                if (mapping.containsKey(neighbor)) {
                    newNeighbor = mapping.get(neighbor);
                }
                else {
                    newNeighbor = new UndirectedGraphNode(neighbor.label);
                    mapping.put(neighbor, newNeighbor);
                    queue.offer(neighbor);
                }
                newNode.neighbors.add(newNeighbor);
            }
        }
        return mapping.get(node);
    }
}
```

劝分不劝合

要把大象装冰箱，总共分几步？

第一步：把冰箱门打开

第二步：把大象装进去

第三步：把冰箱门关上

吃饭和出恭可以一起，但是不香，分开做更好

更好的实现方法

将整个算法分解为三个步骤：

1. 找到所有点
2. 复制所有点
3. 复制所有边

```
def cloneGraph(self, node):
    if not node:
        return None

    # step 1: find nodes
    nodes = self.find_nodes_by_bfs(node)
    # step 2: copy nodes
    mapping = self.copy_nodes(nodes)
    # step 3: copy edges
    self.copy_edges(nodes, mapping)

    return mapping[node]
```

```
def find_nodes_by_bfs(self, node):
    queue = collections.deque([node])
    visited = set([node])
    while queue:
        curt_node = queue.popleft()
        for neighbor in curt_node.neighbors:
            if neighbor in visited:
                continue
            visited.add(neighbor)
            queue.append(neighbor)
    return list(visited)

def copy_nodes(self, nodes):
    mapping = {}
    for node in nodes:
        mapping[node] = UndirectedGraphNode(node.label)
    return mapping

def copy_edges(self, nodes, mapping):
    for node in nodes:
        new_node = mapping[node]
        for neighbor in node.neighbors:
            new_neighbor = mapping[neighbor]
            new_node.neighbors.append(new_neighbor)
```


将整个算法分解为三个步骤:

1. 找到所有点
2. 复制所有点
3. 复制所有边

```
public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
    if (node == null) {
        return null;
    }
    // step 1: find nodes
    List<UndirectedGraphNode> nodes = findNodesByBFS(node);
    // step 2: copy nodes
    Map<UndirectedGraphNode, UndirectedGraphNode> mapping = copyNodes(nodes);
    // step 3: copy edges
    copyEdges(nodes, mapping);
    return mapping.get(node);
}

List<UndirectedGraphNode> findNodesByBFS(UndirectedGraphNode node) {
    Queue<UndirectedGraphNode> queue = new ArrayDeque<>();
    Set<UndirectedGraphNode> visited = new HashSet<>();
    queue.offer(node);
    visited.add(node);
    while (!queue.isEmpty()) {
        UndirectedGraphNode cUndirectedGraphNode = queue.poll();
        for (UndirectedGraphNode neighbor : cUndirectedGraphNode.neighbors) {
            if (visited.contains(neighbor)) {
                continue;
            }
            visited.add(neighbor);
            queue.offer(neighbor);
        }
    }
    return new LinkedList<>(visited);
}
```

更好的实现方法(Java Part 2)

将整个算法分解为三个步骤:

1. 找到所有点
2. 复制所有点
3. 复制所有边

```
Map<UndirectedGraphNode, UndirectedGraphNode> copyNodes(List<UndirectedGraphNode> nodes) {  
    Map<UndirectedGraphNode, UndirectedGraphNode> mapping = new HashMap<>();  
    for (UndirectedGraphNode node : nodes) {  
        mapping.put(node, new UndirectedGraphNode(node.label));  
    }  
    return mapping;  
}  
  
void copyEdges(List<UndirectedGraphNode> nodes, Map<UndirectedGraphNode, UndirectedGraphNode> mapping) {  
    for (UndirectedGraphNode node : nodes) {  
        UndirectedGraphNode newNode = mapping.get(node);  
        for (UndirectedGraphNode neighbor : node.neighbors) {  
            UndirectedGraphNode newNeighbor = mapping.get(neighbor);  
            newNode.neighbors.add(newNeighbor);  
        }  
    }  
}
```


80% 的人都可能会写错的 BFS 算法

- 应该在哪里做访问标记？

```
queue = collections.deque([node])
visited = set([node])
while queue:
    curt_node = queue.popleft()
    for neighbor in curt_node.neighbors:
        if neighbor in visited:
            continue
        visited.add(neighbor)
        queue.append(neighbor)
return list(visited)
```

```
queue = collections.deque([node])
# visited = set([node])
while queue:
    curt_node = queue.popleft()
    visited.add(curt_node)
    for neighbor in curt_node.neighbors:
        if neighbor in visited:
            continue
        # visited.add(neighbor)
        queue.append(neighbor)
return list(visited)
```

```
queue.offer(node);
set.add(node);
while (!queue.isEmpty()) {
    UndirectedGraphNode head = queue.poll();
    for (UndirectedGraphNode neighbor : head.neighbors) {
        if (!set.contains(neighbor)) {
            set.add(neighbor);
            queue.offer(neighbor);
        }
    }
}
```

```
queue.offer(node);
// set.add(node);
while (!queue.isEmpty()) {
    UndirectedGraphNode head = queue.poll();
    set.add(head);
    for (UndirectedGraphNode neighbor : head.neighbors) {
        if (!set.contains(neighbor)) {
            // set.add(neighbor);
            queue.offer(neighbor);
        }
    }
}
```

分层 vs 不分层

代码实现上有什么区别？

```
def find_nodes_by_bfs(self, node):
    queue = collections.deque([node])
    visited = set([node])
    while queue:
        curt_node = queue.popleft()
        for neighbor in curt_node.neighbors:
            if neighbor in visited:
                continue
            visited.add(neighbor)
            queue.append(neighbor)
    return list(visited)
```

```
def find_nodes_by_bfs(self, node):
    queue = collections.deque([node])
    visited = set([node])
    while queue:
        for _ in range(len(queue)):
            curt_node = queue.popleft()
            for neighbor in curt_node.neighbors:
                if neighbor in visited:
                    continue
                visited.add(neighbor)
                queue.append(neighbor)
    return list(visited)
```

```
private ArrayList<UndirectedGraphNode> getNodes(UndirectedGraphNode node) {
    Queue<UndirectedGraphNode> queue = new ArrayDeque<UndirectedGraphNode>();
    HashSet<UndirectedGraphNode> set = new HashSet<>();

    queue.offer(node);
    set.add(node);
    while (!queue.isEmpty()) {
        UndirectedGraphNode head = queue.poll();
        for (UndirectedGraphNode neighbor : head.neighbors) {
            if (!set.contains(neighbor)) {
                set.add(neighbor);
                queue.offer(neighbor);
            }
        }
    }

    return new ArrayList<UndirectedGraphNode>(set);
}
```

```
private ArrayList<UndirectedGraphNode> getNodes(UndirectedGraphNode node) {
    Queue<UndirectedGraphNode> queue = new ArrayDeque<UndirectedGraphNode>();
    HashSet<UndirectedGraphNode> set = new HashSet<>();

    queue.offer(node);
    set.add(node);
    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            UndirectedGraphNode head = queue.poll();
            for (UndirectedGraphNode neighbor : head.neighbors) {
                if (!set.contains(neighbor)) {
                    set.add(neighbor);
                    queue.offer(neighbor);
                }
            }
        }
    }

    return new ArrayList<UndirectedGraphNode>(set);
}
```

Word Ladder

<http://www.lintcode.com/problem/word-ladder/>

<http://www.jiuzhang.com/solution/word-ladder/>

简单图最短路径

给出两个单词（**start**和**end**）和一个字典，找出从**start**到**end**的最短转换序列，输出最短序列的长度。


```
Queue<Node> queue = new ArrayDeque<>();
HashMap<Node, Integer> distance = new HashMap<>();

// step 1: 初始化
// 把初始节点放到 queue 里, 如果有多个就都放进去
// 并标记初始节点的距离为0, 记录在 distance 的 hashmap 里
// distance 有两个作用, 一是判断是否已经访问过, 二是记录离起点的距离
queue.offer(node);
distance.put(node, 0);

// step 2: 不断访问队列
// while 循环 + 每次 pop 队列中的一个点出来
while (!queue.isEmpty()) {
    Node node = queue.poll();
    // step 3: 拓展相邻节点
    // pop 出的节点的相邻节点, 加入队列并在 distance 中存储距离
    for (Node neighbor : node.getNeighbors()) {
        if (distance.containsKey(neighbor)) {
            continue;
        }
        distance.put(neighbor, distance.get(node) + 1);
        queue.offer(neighbor);
    }
}
```

```
# step 1 初始化
# 把初始节点放到 deque 里, 如果有多个就都放进去
# 并标记初始节点的距离为0, 记录在 distance 的 dict 里
# distance 有两个作用, 一是判断是否已经访问过, 二是记录节点距离
queue = collections.deque([node])
distance = {node : 0}

# step 2: 不断访问队列
# while 循环 + 每次 pop 队列中的一个点出来
while queue:
    node = queue.popleft()
    # step 3: 拓展相邻节点
    # pop 出的节点的相邻节点, 加入队列并在 distance 中存储距离
    for neighbor in node.get_neighbors():
        if neighbor in distance:
            continue
        distance[neighbor] = distance[node] + 1
        queue.append(neighbor)
```

休息，休息一会儿



矩阵中的宽度优先搜索

BFS in Matrix

两个优化建议

Python 队列建议使用 `deque` 不建议使用 `Queue`

（涉及多线程加锁会更慢）

Java 队列建议 `new ArrayDeque` 不建议 `new LinkedList`

（链表比数组慢）

图 Graph

N个点，M条边

M最大是 $O(N^2)$ 的级别

图上BFS时间复杂度 = $O(N + M)$

- 说是 $O(M)$ 问题也不大，因为M一般都比N大
所以最坏情况可能是 $O(N^2)$

矩阵 Matrix

R行C列

$R \times C$ 个点， $R \times C \times 2$ 条边（每个点上下左右4条边，每条边被2个点共享）。

矩阵中BFS时间复杂度 = $O(R * C)$

Number of Islands

<http://www.lintcode.com/problem/number-of-islands/>

<http://www.jiuzhang.com/solutions/number-of-islands/>

连通块问题

给一个 01 矩阵，求1构成的联通块的个数。

坐标变换数组

```
int[] deltaX = {1,0,0,-1};
```

```
int[] deltaY = {0,1,-1,0};
```

问：写出八个方向的坐标变换数组？

```
def numIslands(self, grid):
    if not grid or not grid[0]:
        return 0

    self.n, self.m = len(grid), len(grid[0])
    self.visited = [[False] * self.m for _ in range(self.n)]

    islands = 0
    for row in range(self.n):
        for col in range(self.m):
            if self.is_island(grid, row, col):
                self.visited[row][col] = True
                self.dfs(grid, row, col)
                islands += 1

    return islands

def is_island(self, grid, x, y):
    if not (0 <= x < self.n and 0 <= y < self.m):
        return False
    if not grid[x][y]:
        return False
    return not self.visited[x][y]
```

```
def dfs(self, grid, x, y):
    dx = [1, 0, -1, 0]
    dy = [0, 1, 0, -1]
    for direction in range(4):
        newx = x + dx[direction]
        newy = y + dy[direction]

        if self.is_island(grid, newx, newy):
            self.visited[newx][newy] = True
            self.dfs(grid, newx, newy)
            # no backtracking
```

DFS 的方法有什么问题？

```
def numIslands(self, grid):
    if not grid or not grid[0]:
        return 0

    islands = 0
    visited = set()
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] and (i, j) not in visited:
                self.bfs(grid, i, j, visited)
                islands += 1

    return islands

def bfs(self, grid, x, y, visited):
    queue = deque([(x, y)])
    visited.add((x, y))
    while queue:
        x, y = queue.popleft()
        for delta_x, delta_y in DIRECTIONS:
            next_x = x + delta_x
            next_y = y + delta_y
            if not self.is_valid(grid, next_x, next_y, visited):
                continue
            queue.append((next_x, next_y))
            visited.add((next_x, next_y))
```

```
def is_valid(self, grid, x, y, visited):
    n, m = len(grid), len(grid[0])
    if not (0 <= x < n and 0 <= y < m):
        return False
    if (x, y) in visited:
        return False
    return grid[x][y]
```

```

public int numIslands(boolean[][] grid) {
    if (grid == null || grid.length == 0 || grid[0].length == 0) {
        return 0;
    }

    int n = grid.length;
    int m = grid[0].length;
    int islands = 0;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (grid[i][j]) {
                makeByBFS(grid, i, j);
                islands++;
            }
        }
    }

    return islands;
}

private boolean inBound(Coordinate coor, boolean[][] grid) {
    int n = grid.length;
    int m = grid[0].length;

    return coor.x >= 0 && coor.x < n && coor.y >= 0 && coor.y < m;
}

private void markByBFS(boolean[][] grid, int x, int y) {
    // magic numbers!
    int[] directionX = {0, 1, -1, 0};
    int[] directionY = {1, 0, 0, -1};

    Queue<Coordinate> queue = new ArrayDeque<>();

    queue.offer(new Coordinate(x, y));
    grid[x][y] = false;

    while (!queue.isEmpty()) {
        Coordinate coor = queue.poll();
        for (int i = 0; i < 4; i++) {
            Coordinate adj = new Coordinate(
                coor.x + directionX[i],
                coor.y + directionY[i]
            );
            if (!inBound(adj, grid)) {
                continue;
            }
            if (grid[adj.x][adj.y]) {
                grid[adj.x][adj.y] = false;
                queue.offer(adj);
            }
        }
    }
}
    
```

更多 Union Find 有关的问题

将在《九章算法面试高频题冲刺班》中讲解

并查集 Union Find

Knight Shortest Path

<http://www.lintcode.com/problem/knight-shortest-path/>

<http://www.jiuzhang.com/solutions/knight-shortest-path/>

简单图最短路径
八个方向坐标变换

```
public static final int[] dx = {1, 1, -1, -1, 2, 2, -2, -2};
public static final int[] dy = {2, -2, 2, -2, 1, -1, 1, -1};

public int shortestPath(boolean[][] grid, Point source, Point destination) {
    if (grid == null || grid.length == 0) {
        return -1;
    }

    Queue<Point> queue = new ArrayDeque<>();
    Map<Integer, Integer> distance = new HashMap();

    int n = grid.length, m = grid[0].length;
    queue.offer(source);
    distance.put(source.x * m + source.y, 0);

    private boolean isValid(int x, int y, boolean[][] grid) {
        if (x < 0 || x >= grid.length || y < 0 || y >= grid[0].length) {
            return false;
        }
        return !grid[x][y];
    }
}
```

```
while (!queue.isEmpty()) {
    Point point = queue.poll();
    if (point.x == destination.x && point.y == destination.y) {
        return distance.get(point.x * m + point.y);
    }

    for (int i = 0; i < 8; i++) {
        int adjX = point.x + dx[i];
        int adjY = point.y + dy[i];
        if (!isValid(adjX, adjY, grid)) {
            continue;
        }
        if (distance.containsKey(adjX * m + adjY)) {
            continue;
        }

        distance.put(adjX * m + adjY, distance.get(point.x * m + point.y) + 1);
        queue.offer(new Point(adjX, adjY));
    }
}

return -1;
}
```

```
def shortestPath(self, grid, source, destination):
    queue = collections.deque([(source.x, source.y)])
    distance = {(source.x, source.y): 0}

    while queue:
        x, y = queue.popleft()
        if (x, y) == (destination.x, destination.y):
            return distance[(x, y)]
        for dx, dy in DIRECTIONS:
            next_x, next_y = x + dx, y + dy
            if (next_x, next_y) in distance:
                continue
            if not self.is_valid(next_x, next_y, grid):
                continue
            distance[(next_x, next_y)] = distance[(x, y)] + 1
            queue.append((next_x, next_y))
    return -1

def is_valid(self, x, y, grid):
    n, m = len(grid), len(grid[0])

    if x < 0 or x >= n or y < 0 or y >= m:
        return False

    return not grid[x][y]
```

拓扑排序 Topological Sorting

几乎每个公司都有一道拓扑排序的面试题！

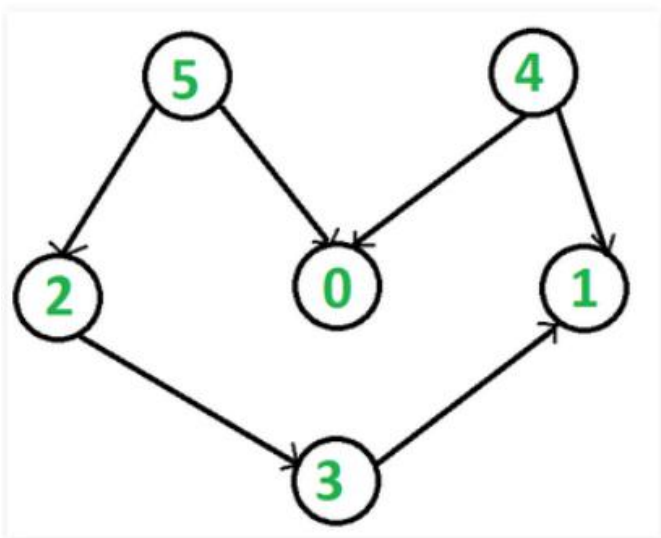
BFS or DFS?

独孤九剑——破剑式

能够用 BFS 解决的问题，一定**不要用** DFS 去做！
因为用 Recursion 实现的 DFS 可能造成 StackOverflow!
(Iteration 的 DFS 一来你不会写，二来面试官也看不懂)

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



Topological Sorting vs Depth First Traversal (DFS):

In **DFS**, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike **DFS**, the vertex '4' should also be printed before vertex '0'. So Topological sorting is different from DFS. For example, a DFS of the shown graph is "5 2 3 1 0 4", but it is not a topological sorting

入度 (In-degree) :

有向图 (Directed Graph) 中指向当前节点的点的个数 (或指向当前节点的边的条数)

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 (Queue) 中作为起始节点
3. 不断从队列中拿出一个点, 去掉这个点的所有连边 (指向其他点的边), 其他点的相应的入度 - 1
4. 一旦发现新的入度为 0 的点, 丢回队列中

拓扑排序并不是传统的排序算法

一个图可能存在多个拓扑序 (Topological Order), 也可能不存在任何拓扑序

拓扑排序的四种不同问法

求任意一个拓扑序

问是否存在拓扑序

求是否存在且仅存在一个拓扑序

求字典序最小的拓扑排序

求任意一个拓扑排序

<http://www.lintcode.com/problem/topological-sorting/>

<http://www.lintcode.com/problem/course-schedule/>

一个个把点从图中抠出来

```
def topSort(self, graph):
    node_to_indegree = self.get_indegree(graph)

    # bfs
    order = []
    start_nodes = [n for n in graph if node_to_indegree[n] == 0]
    queue = collections.deque(start_nodes)
    while queue:
        node = queue.popleft()
        order.append(node)
        for neighbor in node.neighbors:
            node_to_indegree[neighbor] -= 1
            if node_to_indegree[neighbor] == 0:
                queue.append(neighbor)

    return order

def get_indegree(self, graph):
    node_to_indegree = {x: 0 for x in graph}

    for node in graph:
        for neighbor in node.neighbors:
            node_to_indegree[neighbor] += 1

    return node_to_indegree
```

```
public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode> graph) {
    ArrayList<DirectedGraphNode> result = new ArrayList<DirectedGraphNode>();
    HashMap<DirectedGraphNode, Integer> map = new HashMap<>();
    for (DirectedGraphNode node : graph) {
        for (DirectedGraphNode neighbor : node.neighbors) {
            if (map.containsKey(neighbor)) {
                map.put(neighbor, map.get(neighbor) + 1);
            } else {
                map.put(neighbor, 1);
            }
        }
    }

    Queue<DirectedGraphNode> q = new ArrayDeque<DirectedGraphNode>();
    for (DirectedGraphNode node : graph) {
        if (!map.containsKey(node)) {
            q.offer(node);
            result.add(node);
        }
    }

    while (!q.isEmpty()) {
        DirectedGraphNode node = q.poll();
        for (DirectedGraphNode n : node.neighbors) {
            map.put(n, map.get(n) - 1);
            if (map.get(n) == 0) {
                result.add(n);
                q.offer(n);
            }
        }
    }

    return result;
}
```

判断是否存在拓扑排序

<http://www.lintcode.com/problem/course-schedule-ii/>

<http://www.jiuzhang.com/problem/course-schedule-ii/>

所有节点均能从图中被删除进入拓扑序

```
def findOrder(self, numCourses, prerequisites):
    graph = [[] for i in range(numCourses)]
    in_degree = [0] * numCourses

    # 建图
    for node_in, node_out in prerequisites:
        graph[node_out].append(node_in)
        in_degree[node_in] += 1

    num_choose = 0
    queue = collections.deque()
    topo_order = []

    # 将入度为 0 的编号加入队列
    for i in range(numCourses):
        if in_degree[i] == 0:
            queue.append(i)

    while queue:
        now_pos = queue.popleft()
        topo_order.append(now_pos)
        num_choose += 1
        # 将每条邻边删去, 如果入度降为 0, 再加入队列
        for next_pos in graph[now_pos]:
            in_degree[next_pos] -= 1
            if in_degree[next_pos] == 0:
                queue.append(next_pos)

    if num_choose == numCourses:
        return topo_order
    return []
```

```
public int[] findOrder(int numCourses, int[][] prerequisites) {
    List[] graph = new ArrayList[numCourses];
    int[] inDegree = new int[numCourses];

    for (int i = 0; i < numCourses; i++) {
        graph[i] = new ArrayList<Integer>();
    }

    // 建图
    for (int[] edge: prerequisites) {
        graph[edge[1]].add(edge[0]);
        inDegree[edge[0]]++;
    }

    int numChoose = 0;
    Queue queue = new LinkedList();
    int[] topoOrder = new int[numCourses];

    // 将入度为 0 的编号加入队列
    for (int i = 0; i < inDegree.length; i++) {
        if (inDegree[i] == 0) {
            queue.add(i);
        }
    }
}
```

```
while (!queue.isEmpty()) {
    int nowPos = (int)queue.poll();
    topoOrder[numChoose] = nowPos;
    numChoose++;
    // 将每条边删去, 如果入度降为 0, 再加入队列
    for (int i = 0; i < graph[nowPos].size(); i++) {
        int nextPos = (int)graph[nowPos].get(i);
        inDegree[nextPos]--;
        if (inDegree[nextPos] == 0) {
            queue.add(nextPos);
        }
    }
}

if (numChoose == numCourses) {
    return topoOrder;
}
return new int[0];
}
```

问拓扑序是否唯一

<http://www.lintcode.com/problem/sequence-reconstruction/>

<http://www.jiuzhang.com/problem/sequence-reconstruction/>

保持队列中有且仅有一个元素

```
def sequenceReconstruction(self, org, seqs):
    graph = self.build_graph(seqs)
    topo_order = self.topological_sort(graph)
    return topo_order == org

def build_graph(self, seqs):
    # initialize graph
    graph = {}
    for seq in seqs:
        for node in seq:
            if node not in graph:
                graph[node] = set()

    for seq in seqs:
        for i in range(1, len(seq)):
            graph[seq[i - 1]].add(seq[i])

    return graph

def get_indegrees(self, graph):
    indegrees = {
        node: 0
        for node in graph
    }

    for node in graph:
        for neighbor in graph[node]:
            indegrees[neighbor] += 1

    return indegrees
```

```
def topological_sort(self, graph):
    indegrees = self.get_indegrees(graph)

    queue = []
    for node in graph:
        if indegrees[node] == 0:
            queue.append(node)

    topo_order = []
    while queue:
        if len(queue) > 1:
            # there must exist more than one topo orders
            return None

        node = queue.pop()
        topo_order.append(node)
        for neighbor in graph[node]:
            indegrees[neighbor] -= 1
            if indegrees[neighbor] == 0:
                queue.append(neighbor)

    if len(topo_order) == len(graph):
        return topo_order

    return None
```



```
public boolean sequenceReconstruction(int[] org, int[][] seqs) {
    Map<Integer, Set<Integer>> graph = buildGraph(seqs);
    List<Integer> topoOrder = getTopoOrder(graph);

    if (topoOrder == null || topoOrder.size() != org.length) {
        return false;
    }
    for (int i = 0; i < org.length; i++) {
        if (org[i] != topoOrder.get(i)) {
            return false;
        }
    }
    return true;
}

private Map<Integer, Set<Integer>> buildGraph(int[][] seqs) {
    Map<Integer, Set<Integer>> graph = new HashMap();
    for (int[] seq : seqs) {
        for (int i = 0; i < seq.length; i++) {
            if (!graph.containsKey(seq[i])) {
                graph.put(seq[i], new HashSet<Integer>());
            }
        }
    }
    for (int[] seq : seqs) {
        for (int i = 1; i < seq.length; i++) {
            graph.get(seq[i - 1]).add(seq[i]);
        }
    }
    return graph;
}
```

```
private List<Integer> getTopoOrder(Map<Integer, Set<Integer>> graph) {
    Map<Integer, Integer> indegrees = getIndegrees(graph);
    Queue<Integer> queue = new ArrayDeque();
    List<Integer> topoOrder = new ArrayList();

    for (Integer node : graph.keySet()) {
        if (indegrees.get(node) == 0) {
            queue.offer(node);
            topoOrder.add(node);
        }
    }

    while (!queue.isEmpty()) {
        if (queue.size() > 1) {
            return null;
        }

        Integer node = queue.poll();
        for (Integer neighbor : graph.get(node)) {
            indegrees.put(neighbor, indegrees.get(neighbor) - 1);
            if (indegrees.get(neighbor) == 0) {
                queue.offer(neighbor);
                topoOrder.add(neighbor);
            }
        }
    }

    if (graph.size() == topoOrder.size()) {
        return topoOrder;
    }

    return null;
}
```



```
private Map<Integer, Integer> getIndegrees(Map<Integer, Set<Integer>> graph) {  
    Map<Integer, Integer> indegrees = new HashMap();  
    for (Integer node : graph.keySet()) {  
        indegrees.put(node, 0);  
    }  
    for (Integer node : graph.keySet()) {  
        for (Integer neighbor : graph.get(node)) {  
            indegrees.put(neighbor, indegrees.get(neighbor) + 1);  
        }  
    }  
    return indegrees;  
}
```

求字典序最小的拓扑排序

<http://www.lintcode.com/problem/alien-dictionary/>

<http://www.jiuzhang.com/solution/alien-dictionary/>

应使用什么数据结构？

```
public String alienOrder(String[] words) {  
    Map<Character, Set<Character>> graph = constructGraph(words);  
    if (graph == null) {  
        return "";  
    }  
    return topologicalSorting(graph);  
}
```

```
private Map<Character, Integer> getIndegree(Map<Character, Set<Character>> graph) {  
    Map<Character, Integer> indegree = new HashMap<>();  
    for (Character u : graph.keySet()) {  
        indegree.put(u, 0);  
    }  
  
    for (Character u : graph.keySet()) {  
        for (Character v : graph.get(u)) {  
            indegree.put(v, indegree.get(v) + 1);  
        }  
    }  
  
    return indegree;  
}
```

```
private Map<Character, Set<Character>> constructGraph(String[] words) {  
    Map<Character, Set<Character>> graph = new HashMap<>();  
  
    // create nodes  
    for (int i = 0; i < words.length; i++) {  
        for (int j = 0; j < words[i].length(); j++) {  
            char c = words[i].charAt(j);  
            if (!graph.containsKey(c)) {  
                graph.put(c, new HashSet<Character>());  
            }  
        }  
    }  
  
    // create edges  
    for (int i = 0; i < words.length - 1; i++) {  
        int index = 0;  
        while (index < words[i].length() && index < words[i + 1].length()) {  
            if (words[i].charAt(index) != words[i + 1].charAt(index)) {  
                graph.get(words[i].charAt(index)).add(words[i + 1].charAt(index));  
                break;  
            }  
            index++;  
        }  
        if (index == Math.min(words[i].length(), words[i + 1].length())) {  
            if (words[i].length() > words[i + 1].length()) {  
                return null;  
            }  
        }  
    }  
  
    return graph;  
}
```

```
private String topologicalSorting(Map<Character, Set<Character>> graph) {  
    // as we should return the topo order with lexicographical order  
    // we should use PriorityQueue instead of a FIFO Queue  
    Map<Character, Integer> indegree = getIndegree(graph);  
    Queue<Character> queue = new PriorityQueue<>();  
  
    for (Character u : indegree.keySet()) {  
        if (indegree.get(u) == 0) {  
            queue.offer(u);  
        }  
    }  
  
    StringBuilder sb = new StringBuilder();  
    while (!queue.isEmpty()) {  
        Character head = queue.poll();  
        sb.append(head);  
        for (Character neighbor : graph.get(head)) {  
            indegree.put(neighbor, indegree.get(neighbor) - 1);  
            if (indegree.get(neighbor) == 0) {  
                queue.offer(neighbor);  
            }  
        }  
    }  
    if (sb.length() != indegree.size()) {  
        return "";  
    }  
    return sb.toString();  
}
```

```
def alienOrder(self, words):
    graph = self.build_graph(words)
    if not graph:
        return ""
    return self.topological_sort(graph)

def build_graph(self, words):
    # key is node, value is neighbors
    graph = {}

    # initialize graph
    for word in words:
        for c in word:
            if c not in graph:
                graph[c] = set()

    # add edges
    n = len(words)
    for i in range(n - 1):
        for j in range(min(len(words[i]), len(words[i + 1]))):
            if words[i][j] != words[i + 1][j]:
                graph[words[i][j]].add(words[i + 1][j])
                break
        if j == min(len(words[i]), len(words[i + 1])) - 1:
            if len(words[i]) > len(words[i + 1]):
                return None

    return graph
```

```
def topological_sort(self, graph):
    # initialize indegree
    indegree = {
        node: 0
        for node in graph
    }

    # calculate indegree
    for node in graph:
        for neighbor in graph[node]:
            indegree[neighbor] = indegree[neighbor] + 1

    # use heapq instead of regular queue so that we can get the
    # smallest lexicographical order
    queue = [node for node in graph if indegree[node] == 0]
    heapify(queue)

    # regular bfs algorithm to do topological sorting
    topo_order = ""
    while queue:
        node = heappop(queue)
        topo_order += node
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                heappush(queue, neighbor)

    # if all nodes popped
    if len(topo_order) == len(graph):
        return topo_order

    return ""
```

- 图上的 BFS
 - 判断一个图是否是一棵树
 - <http://www.lintcode.com/problem/graph-valid-tree/>
 - 搜索图中最近值为target的点
 - <http://www.lintcode.com/problem/search-graph-nodes/>
 - 无向图连通块
 - <http://www.lintcode.com/problem/connected-component-in-undirected-graph/>
- 矩阵上的 BFS
 - 僵尸多少天吃掉所有人
 - <http://www.lintcode.com/problem/zombie-in-matrix/>
 - 建邮局问题 Build Post Office II
 - <http://www.lintcode.com/problem/build-post-office-ii/>

- 能用 **BFS** 的一定不要用 **DFS**（除非面试官特别要求）
- **BFS** 的三个使用场景
 - 连通块问题
 - 层级遍历问题
 - 拓扑排序问题
- 是否需要层级遍历
 - 需要多一重循环
 - 或者使用 **distance** 哈希表记录到所有点的距离
- 矩阵坐标变换数组
 - **deltaX, deltaY**
 - **inBound / isValid**

在第 6 周的互动课中继续学习如下与 BFS 有关的内容

- 第三十二章 使用宽度优先搜索找出所有方案
 - 使用 BFS 求所有方案类问题
 - 使用 BFS 序列化二叉树
 - 什么是序列化与反序列化
- 第三十三章 【互动】双向宽度优先搜索算法
 - 双向宽度优先搜索算法
 - 双向宽度优先搜索到底优化了多少
 - 如何优雅的实现双向宽度优先搜索