1 知识点

1.1 基础知识

- 这里的graph都是指simple graph
  - 没有AB，BA
  - 没有AA，自己到自己
- 顶点: vertex, a.k.a. node; 边: edge
  - vertices with an edge between are adjacent
  - Optional: Vertices or edges may have labels (or weights)
- 图的表示
  - 邻接表(**adjacent table**)：每个节点的邻接点的总集合
    - 时间复杂度：O(V + E)
    - 用list表示，例如：[[1],[2],[3]]，0指向1，1指向2，2指向3
    - 用dict表示，例如：{0:[1], 1:[2], 2:[3], 3:[]}
  - 邻接顶点
    - 无向图：由一条边连接起来的两个顶点，互为邻接顶点

```
for i, j in edges:
    graph[j].append(i)
    graph[i].append(j)
```

    - 有向图：u->v, u的邻接点是v，即指向的点为邻接点

```
for i, j in edges:
    graph[j].append(i)
```

  - Adjacency Matrix（领接表）
    - 时间复杂度V^2

- 有环无环，单个多个

n个顶点，edges，无平行边

- 无环孤立
  - n == 1，len(edges)==0
  - n > 1 --> n = len(edges) + 1
- 有环孤立
  - 可以是多个环
  - n < len(edges) + 1
- 无环多个

    - n > len(edges) + 1
- 有环多个
    - n >=< len(edges) + 1

1.2 常见题型

诀窍
用BFS时，append之后立刻visited = 1，以防重复出错

1.2.1 无向图

- 判断两点之间是否相连（遍历）
- 是否有环
    - DFS: visited(0,1)；parent法；核心：无环的时候，其访问过的节点有且只有一个，并且是其上一个节点
    - BFS: 同DFS
    - Union find: root1 == root2则有环
- 孤立个数
    - DFS
    - BFS
    - Union find: n(节点个数) - numb(连接的次数）
- Biconnectivity
    - Is there a vertex whose removal disconnects the graph?
- 路径层数
- 路径和
- 最短路径
- 最小生成树

1.2.2 有向图

- 判断a点能否到达b点（遍历）
- 是否有环
    - DFS: visited(-1, 0, 1)；核心：遇到正在被访问的node(visited[i] == 0)，则有环
    - Topo sort: indegree / outdegree; 遍历完之后，有入度不为0的node，则有环
- 孤立个数
    - DFS / BFS
- 路径层数
- 路径和
- 最短路径
- 最长路径

1.3 时间空间复杂度

- DFS
- BFS

  时间空间都是 O(E+V)
  空间：

  - 建立visited，用V
  - 建立graph，用E
  - DFS：会有保存的栈，也就是考虑最大层数，最坏情况是有V层
  - BFS：会有保存的栈，也就是考虑最宽的层，最坏情况是E

  时间：

  - 建立visited，V
  - 建立graph，V
  - 遍历edges，E
  - DFS/BFS遍历图，V
- 并查集
- 拓扑

2 题目

2.1 图的基础

- 133.Clone Graph

2.2 无向图

判断孤立和分群

- 323.Number of Connected Components in an Undirected Graph(并查集)
- 547.Friend Circles(并查集)
- 1202.Smallest String With Swapy

判断环

- 261.Graph Valid Tree

路径和-求层数

- 无向图的最长距离.py

2.3 有向图

判断孤立个数和分群

- 841.Keys and Rooms

判断环

- 207.Course Schedule
- 802.Find Eventual Safe States

路径和-求层数

- 1376.Time Needed to Inform All Employees
- 210.Course Schedule II
- 399.Evaluate Division
- 444.Sequence Reconstruction
- 程序的依赖关系.py
- 1376 通知所有员工所需的时间

2.4 网格搜索

- 200.Number of Islands
- 733.Flood Fill

2.5 笔记

- 无向图遍历-BFS.py
- 无向图遍历-DFS.py
- 无向图判断是否有环.py
- 有向图判断是否有环.py
- 拓扑排序.py
- 并查集.py

未完成

- 无向图的最长距离

图的建立与应用 565
深度优先搜索 17、397
回溯法　526、401、36、37、51、52、77、39、216、40、46、47、31、556、60、491、78、90、79、93、332
回溯法与表达式 241、282、679
回溯法与括号 22、301
回溯法与贪心 488
广度优先搜索 133、200、695、463、542、130、417、529、127、126、433、675
并查集　547、684、685
拓扑排序 399、207、210
有限状态自动机 65、468

代码总结

3.1 无向图

3.1.1 遍历

3.1.1.1 DepthFirstPath

- 深度优先
  - preorder
  - postorder

In python

```
# 如果不是从0开始的node
from collections import defaultdict
pairs = [[1,2],[3,2]]
graph = defaultdict(list)
for i, j in pairs:
    graph[i].append(j)
graph
```

```
class Graph:
    def DFS_Traversal(self, nums, pairs):
        def dfs(node):
            if visited[node] == 1:
                return

            preorder.append(node)  # 前序遍历
            visited[node] = 1
            for out in adjacent[node]:
                dfs(out)

            postorder.append(node)  # 后序遍历
            return

        # corner case
        if nums == 0 or pairs == []:
            return

        # make visited list and adjacent list
        visited = [0] * nums
        adjacent = [[] for _ in range(nums)]
        for i, j in pairs:
            adjacent[i].append(j)
            adjacent[j].append(i)

        # traversal the graph
```

```
        preorder = []
        postorder = []
        for i in range(nums):
            if visited[i] == 0:
                dfs(i)

        print(preorder)
        print(postorder)
        return

x = Graph()
x.DFS_Traversal(5, [[0,1], [0,2], [0,3], [1,4]])
```

In java

```
import java.util.List;
import java.util.ArrayList;

class Graph {
    int[] visited;
    List<List<Integer>> adjacent;
    ArrayList<Integer> postorder = new ArrayList<Integer>();
    ArrayList<Integer> preorder = new ArrayList<Integer>();

    public void DFStraversal(int nums, int[][] pairs) {
        if (nums == 0 || pairs.length == 0) return;

        visited = new int[nums];
        adjacent = new ArrayList<List<Integer>>();
        for (int i = 0; i < nums; i++) {
            adjacent.add(new ArrayList<Integer>());
        }

        for (int[] each: pairs) {
            adjacent.get(each[0]).add(each[1]);
            adjacent.get(each[1]).add(each[0]);
        }

        for (int i = 0; i< nums; i ++) {
            if (visited[i] == 0) {
                dfs(i);
            }
        }
        System.out.println(preorder);
        System.out.println(postorder);
        return;
```

```
    }

    /** traversal the graph*/
    private void dfs(int node) {
        if (visited[node] == 1) return;

        visited[node] = 1;
        preorder.add(node);
        for (int out: adjacent.get(node)) {
            dfs(out);
        }
        postorder.add(node);
        return;
    }

    public static void main(String[] args) {
        int nums = 5;
        int[][] pairs = {{0,1}, {0,2}, {0,3}, {1,4}};
        Graph graph = new Graph();
        graph.DFStraversal(nums, pairs);

    }
}
```

3.1.1.2 BreadthFirstSearch

- 广度优先

In python

```
from collections import deque

class Graph:
    def BFS_Traversal(self, nums, pairs):
        # corner case
        if nums == 0 or pairs == []:
            return

        visited = [0] * nums
        adjacent = [[] for _ in range(nums)]
        for i, j in pairs:
            adjacent[i].append(j)
            adjacent[j].append(i)

        order = []
        queue = deque()
        for i in range(nums):
```

```
            if visited[i] == 0:
                queue.append(i)
                visited[i] = 1  # 切记，append之后立刻visited变成1

                while queue:
                    node = queue.popleft()
                    order.append(node)

                    for out in adjacent[node]:
                        if visited[out] == 0:
                            queue.append(out)
                            visited[out] = 1
        print(order)
        return

x = Graph()
x.BFS_Traversal(5, [[0,1], [0,2], [0,3], [1,4]])
```

In java

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

class Graph {
    int[] visited;
    List<List<Integer>> adjacent;
    ArrayList<Integer> order = new ArrayList<Integer>();

    public void BFStraversal(int nums, int[][] pairs) {
        if (nums == 0 || pairs.length == 0) return;

        visited = new int[nums];
        adjacent = new ArrayList<List<Integer>>();
        for (int i = 0; i < nums; i++) {
            adjacent.add(new ArrayList<Integer>());
        }

        for (int[] each: pairs) {
            adjacent.get(each[0]).add(each[1]);
            adjacent.get(each[1]).add(each[0]);
        }

        Queue<Integer> queue = new LinkedList<Integer>();
        for (int i = 0; i < nums; i++) {
```

```
            if (visited[i] == 0) {
                queue.offer(i);
                visited[i] = 1;

                while (!queue.isEmpty()) {
                    int node = queue.poll();
                    order.add(node);
                    System.out.println(node);

                    for (int out : adjacent.get(node)) {
                        if (visited[out] == 0) {
                            queue.offer(out);
                            visited[out] = 1;
                        }
                    }
                }
            }
        }
        System.out.println(order);
        return;
    }

    public static void main(String[] args) {
        int nums = 5;
        int[][] pairs = {{0,1}, {0,2}, {0,3}, {1,4}};
        Graph graph = new Graph();
        graph.BFStraversal(nums, pairs);
    }
}
```

3.1.2 判断环

3.1.2.1 DFS的01，parent法则

- 用visited来记录是否已经被访问过（0，1法则）
- 同时记录该节点的parent node
- **无环：其访问过的节点有且只有一个，并且是其节点的上一个节点（parent_index)**

In python

```
# True: 有环
# False: 没有环
# 无环：其访问过的节点有且只有一个，并且是其节点的上一个节点（parent_index)

class Graph:
```

```
    def findCircle(self, nums, pairs):
        if nums == 0 or pairs == [] or pairs == [[]]:
            return False
        '''
        判断是否无圈且是一个component
        if n == 0:
            return False
        # n=2, pairs = [] --> 多个component
        if n - 1 != len(edges):
            return True
        '''
        visited = [0] * nums
        graph = [[] for _ in range(nums)]

        for i, j in pairs:
            graph[i].append(j)
            graph[j].append(i)

        def dfs(index, parent):
            visited[index] = 1

            for j in graph[index]:
                if visited[j] == 0:
                    if dfs(j, index) == True:
                        return True
                elif j != parent:
                    return True
            return False

        for i in range[nums]:
            if visited[i] == 0:
                if dfs(i, -1) == True:
                    return True
        return False
```

In java

```
/** True: 有环*/
private Boolean dfs(int node, int parent) {
    visited[node] = 1;
    for (int out: adjacent.get(node)) {
        if (visited[out] == 0) {
            if (dfs(out, node) == true) {
                return true;
            }
        } else {
```

```
            if (out != parent) {
                return true;
            }
        }
    }
    return false;
}
```

3.1.2.2 BFS的01，parent法则

In python

```python
# True: 有环
# False: 没有环

from collections import deque, defaultdict
class Graph:
    def findCircle(self, nums, pairs):
        if nums == None or pairs == None:
            return False

        visited = [0] * nums
        graph = defaultdict(set)
        queue = deque()

        for i, j in pairs:
            graph[i].add(j)
            graph[j].add(i)

        for i in range(nums):
            if visited[i] == 0:
                queue.append([i,-1])

            while queue:
                index, parent = queue.popleft()
                visited[index] = 1

                for out_index in graph[index]:
                    if visited[out_index] == 0:
                        queue.append([out_index, index])
                    elif out_index != parent:
                        return False
        return True
```

In java

```

```

3.1.2.3 Union Find

- Union find by rank and path compression
- if root1 == root2 则有环

In python

```python
# True: 有环
# False: 没有环

class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        if n == 0:
            return 0

        if len(edges) == 0:
            return n

        def find(i):
            if gT[i] == i:
                return i
            else:
                return find(gT[i])

        count = 0
        gT = [x for x in range(n)]
        rank = [1] * n
        for i, j in edges:
            root1 = find(i)
            root2 = find(j)

            if root1 == root2:
                return True
            else:
                if rank[i] >= rank[j]:
                    rank[i] += rank[j]
                    gT[root2] = root1
                else:
                    rank[j] += rank[i]
                    gT[root1] = root2
        return False
```

In java

```
// union find 见岛屿个数处
```

3.1.3 孤立岛屿个数

- DFS 0 1 法则
- BFS 0 1 法则
- Union find

In python

```
class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        if n == 0:
            return 0

        if len(edges) == 0:
            return n

        def find(i):
            if gT[i] == i:
                return i
            else:
                return find(gT[i])

        count = 0
        gT = [x for x in range(n)]
        for i, j in edges:
            root1 = find(i)
            root2 = find(j)

            if root1 == root2:
                pass
            else:
                gT[root2] = root1
                count += 1
        return n - count
```

In java

```
class Solution {
    private int[] gT;
    private int[] size;
```

```java
    public int countComponents(int n, int[][] edges) {
        gT = new int[n];
        size = new int[n];
        int count = n;

        for (int i = 0; i < n; i += 1) {
            gT[i] = i;
            size[i] = 1;
        }

        for (int [] e : edges) {
            int root1 = find(e[0]);
            int root2 = find(e[1]);
            if (root1 == root2) continue;

            if (size[root1] >= size[root2]) {
                gT[root2] = root1;
                size[root1] += size[root2];
            } else {
                gT[root1] = root2;
                size[root2] += size[root1];
            }

            count -= 1;
        }

        for (int i = 0; i < n; i++) {
            if (gT[i] == i) {
                System.out.print(size[i]);
            }
        }

        return count;
    }

    private int find(int i) {
        while (gT[i] != i) {
            gT[i] = gT[gT[i]];
            i = gT[i];
        }
        return i;
    }
}
```

3.1.4 路径类

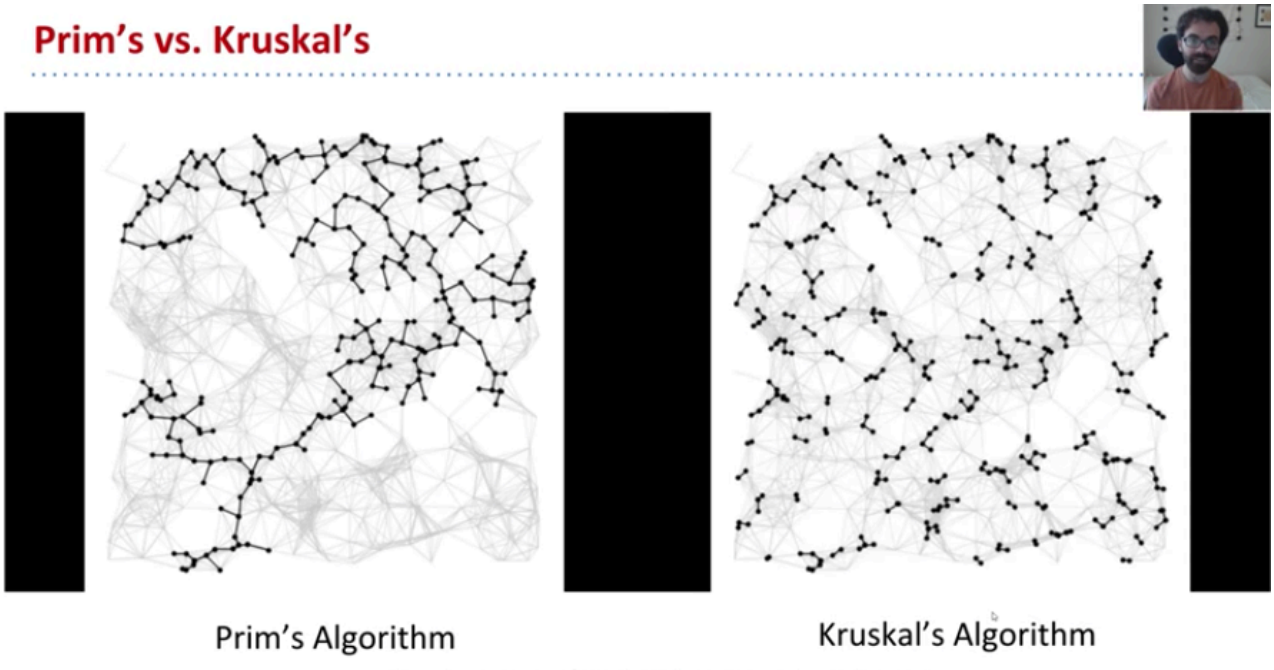
3.1.4.1 所有路径

3.1.4.2 最短路径

`Dijkstra` 算法？

3.1.4.3 最小生成树(MST)

- Minimum Spanning Trees
- acyclic
- 只考虑the graph is connected, 如果有多个component，分别求最小生成树，然后合并在一起
- 如果不同边的权重可以相同，那么最小生成树就不是唯一性的了，当所有边的权重都各不相同时，就只有一个最小生成树
- 1584



- prim是从一点出发，每一点都是连在一起的
- kruskal是一块块来，最后都连在一起

Prim's Algorithm

思路

- 关键点在于每次都找两点之间的最短，把所有点连接起来就是最小生成树
- 把所有点以inf的起始放入

- 随意找一个起始点，起始权重是0
- 每一次pop最短的那个点
- 遍历其相邻点，改变path， 这里的path是指两点之间的最短权重
    - Dijkstra里面的path是source点到该点之间的最短权重
- 注意已经找到最短的点就不要再继续找了

```
# 方法一，非heap
# 按点为基准，遍历点
def Prim_findMST(self, n, edges):
    adj = defaultdict(dict)
    for i, j, w in edges:
        adj[i][j] = w
        adj[j][i] = w

    count = 0
    res = {}
    dist = {node: float('inf') for node in range(n)}
    dist[0] = 0

    while dist:
        i, w = sorted(dist.items(), key = lambda x:x[1])[0]
        dist.pop(i)

        res[i] = w
        count += w

        for j in adj[i]:
            if j not in res:
                dist[j] = min(dist[j], adj[i][j])

    return count

# 方法二
# heap里面放边，遍历边
def findMST(self, n: int, edges: List[List[int]]):  # node: 0-start
    # adjancent list
    adj = defaultdict(dict)

    for i, j, w in edges:
        adj[i][j] = w
        adj[j][i] = w

    heap = [(0, 0)]  # path, node
    res = {}
    count = 0

    while heap:
```

```
        w, i = heappop(heap)

        if i not in res:
            count += w
            res[i] = w

        if len(res) == n:
            break

        for j in adj[i]:
            if j not in res:
                heappush(heap, (adj[i][j], j))

    return count
```

Kruskal's Algorithm

- insert all edges into PQ
- Repeat: Remove smallest weight edge. Add to MST if no cycle created
    - check cycle: Union(node1, node2)

```
def kruskal(pair, n):
    def find(v):
        while groupTag[v] != v:
            groupTag[v] = groupTag[groupTag[v]]
            v = groupTag[v]
        return v

    def union(root1, root2):
        if rank[root1] < rank[root2]:
            root1, root2 = root2, root1
        rank[root1] += rank[root2]
        groupTag[root2] = root1

        return

    heap = [(w, i, j) for i, j, w in pair]
    heapify(heap)

    res = 0
    edgeTime = 0
    groupTag = {i:i for i in range(n)}
    rank = {i : 1 for i in range(n)}

    while edgeTime != n - 1:  # 前提是所有的点都在一个component上
```

```
        w, i, j = heappop(heap)
        root1 = find(i)
        root2 = find(j)
        if root1 != root2:
            union(root1, root2)
            res += w
            edgeTime += 1
    return res
```

3.2 有向图

3.2.1 遍历

- 构建连接表的时候
  - 无向图：由一条边连接起来的两个顶点，互为邻接顶点

    ```
    for i, j in edges:
        graph[j].append(i)
        graph[i].append(j)
    ```

  - 有向图：u->v, u的邻接点是v，即指向的点为邻接点

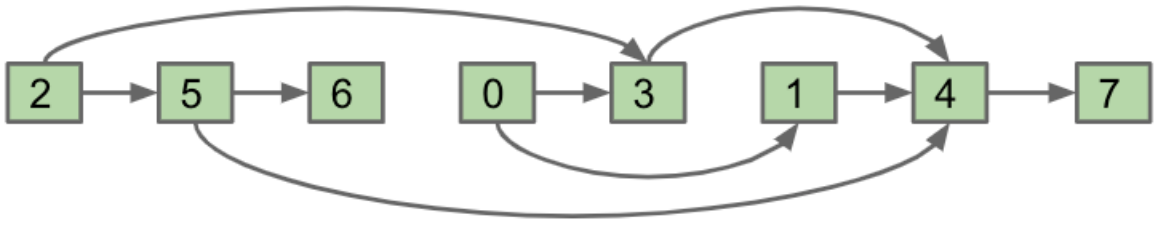
    ```
    for i, j in edges:
        graph[j].append(i)
    ```

- DFS
- BFS

3.2.1.1 Topological Sort

广度优先

- **topological ordering is given by the reverse of that list(reverse postorder)**
  - `[2, 5, 6, 0, 3, 1, 4, 7]`
  - 把这些nodes都整理一下就是下面的图
  - 遍历方向都是朝着右边的



```python
# 本质就是DFS [-1,0,1]的后序遍历，然后后序遍历的结果倒置就是topo sort
# 210

'''
b --> a
'''

class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) ->
List[int]:
        def DFS(i):
            '''
            True: no circle
            False: have circle
            '''
            if visited[i] == 1:
                return True
            if visited[i] == 0:
                return False

            visited[i] = 0
            for out in adj[i]:
                if DFS(out) == False:
                    return False
            visited[i] = 1
            self.res.append(i)
            return True

        p = prerequisites
        if p == [] or p == [[]]:
            return [x for x in range(numCourses)]

        visited = [-1] * numCourses
        adj = [[] for _ in range(numCourses)]

        for i, j in p:
            adj[j].append(i)

        self.res = []
        for i in range(numCourses):
            if visited[i] == -1:
                if DFS(i) == False:
                    return []
        return self.res[::-1]
```

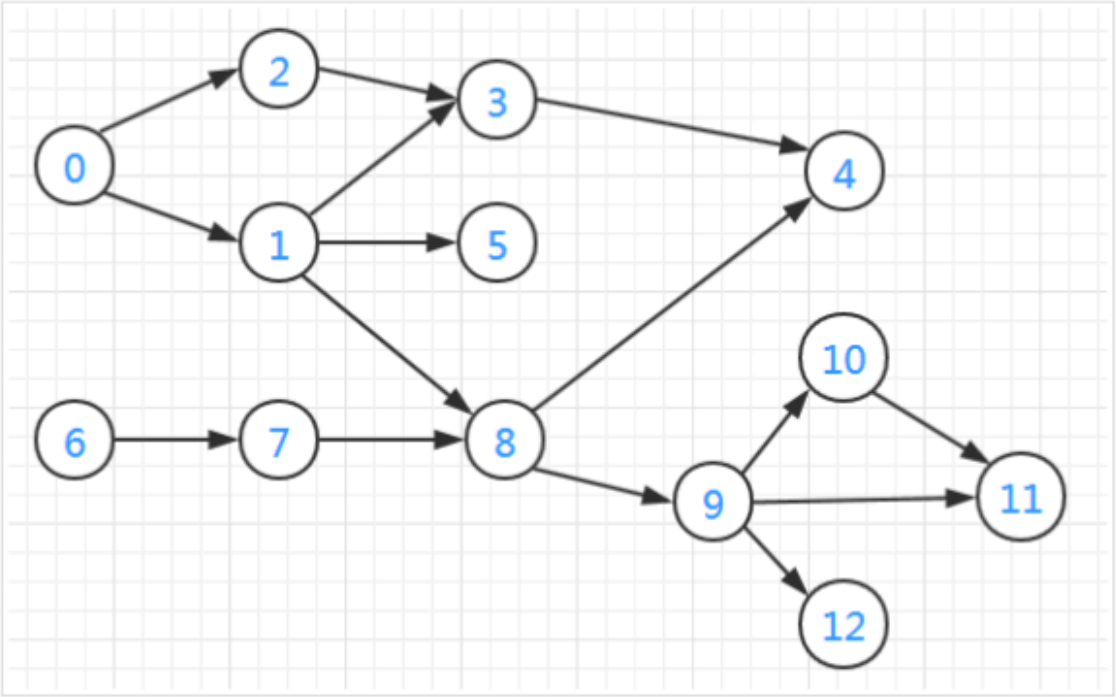
宽度优先

Solution

1. Calculate in-degree of all vertices.
2. Pick any vertex v which has in-degree of 0.
3. Add v to our topological sort list. Remove the vertex v and all edges coming out of it. Decrement in-degrees of all neighbors of vertex v by 1.
4. Repeat steps 2 and 3 until all vertices are removed.

```
from collections import deque
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) ->
List[int]:
        p = prerequisites
        if p == [] or p == [[]]:
            return [x for x in range(numCourses)]

        indegree = [0] * numCourses
        adj = [[] for _ in range(numCourses)]

        for i, j in p:
            # j --> i
            adj[j].append(i)
            indegree[i] += 1

        queue = deque()
        res = []
        for i in range(numCourses):
            if indegree[i] == 0:
                queue.append(i)

        while queue:
            node = queue.popleft()
            res.append(node)

            for out in adj[node]:
                indegree[out] -= 1
                if indegree[out] == 0:
                    queue.append(out)

        return res if sum(indegree) == 0 else []
```

两种实现方式结果对比



- 深度优先搜索

  拓扑排序结果：6, 7, 0, 2, 1, 8, 9, 12, 10, 11, 5, 3, 4 ， 体现了排序纵深特性。
- 广度优先搜索

  拓扑排序结果：0, 6, 1, 2, 7, 5, 3, 8, 9, 4, 10, 12, 11 ， 体现了排序分层特性。

时间复杂度

不管是广度还是深度优先，实现拓扑排序的时间复杂度都为 `O(V+E)` 。

3.2.2 判断环

3.2.2.1 DFS [-1,0,1]法则

- 遇到正在被访问的点，则说明有环
- 参考遍历里的DFS

3.2.2.2 拓扑

- 最后indgree里面还有不是0的节点，则有环
- 参考遍历里的topo 广度优先

3.2.3 孤立岛屿个数

- 模版同无向图

3.2.4 路径类

- 210.Course Schedule II
- 399.Evaluate Division
- 444.Sequence Reconstruction
- 1376.Time Needed to Inform All Employees
- 程序的依赖关系.py

3.2.4.1 路径和/层数

- 依赖关系
- 课程表
    - 拓扑排序

3.2.4.2 单点最短路径

`Dijkstra` 算法

- 特点：支持环，仅能处理权重**全为正**的有向图 （无向图是否可以使用？？？）
- 时间复杂度为 `O(ElogV)`
- 边的放松顺序
  小根堆取出距离起点距离最近的顶点，依次放松该顶点指向邻接点的边；且每条边只会被放松一次。

`DAG` 有向无环图算法

- DAG SPT Algorithm
    - Directed Acyclic Graphs Shortest Path
- 特点：不支持环，权重**正或负**都能处理
- 时间复杂度为 `O(E+V)`
- 边的放松顺序
  按照顶点的拓扑顺序，放松所有的边；且所有的边只会被放松一次。

Dijkstra

- 迪杰斯特拉算法
- source点到各个node的最短
- 不是综合最短！！！
- 743题(带环)

思路1 主要记这个

1. 把所有点以inf的起始放入
2. source点是0
3. 每一次pop最短的那个点
4. 遍历其相邻点，改变path
    - 注意这里的path指的是source点到该点之间的最短权重
    - 优化版的Prim里的path指的是两点之间的最短权重
5. 时间复杂度： O(V*VlogV) 空间复杂度: O(V)

```
# 基本模型
# 先把所有点都加入到dict里
# 每次pop最近的一个点
# 以点为基准
# from CS61B

from collections import defaultdict

class Solution:
    def networkDelayTime(self, times: List[List[int]], N: int, K: int) -> int:
        # adjancent list
        adj = defaultdict(dict)
        for i, j, w in times:
            adj[i][j] = w

        # init a dist
        dist = {node: float('inf') for node in range(1, N + 1)}
        dist[K] = 0

        # find the shortest paths
        res = {}
        while dist:
            i, w = sorted(dist.items(), key = lambda x:x[1])[0]
            res[i] = w
            dist.pop(i)

            for j in adj[i]:
                if j not in res:
                    dist[j] = min(dist[j], w + adj[i][j])
        res = sorted(res.values(), key = lambda x:x)[-1]
        return res if res != float('inf') else -1
```

思路2

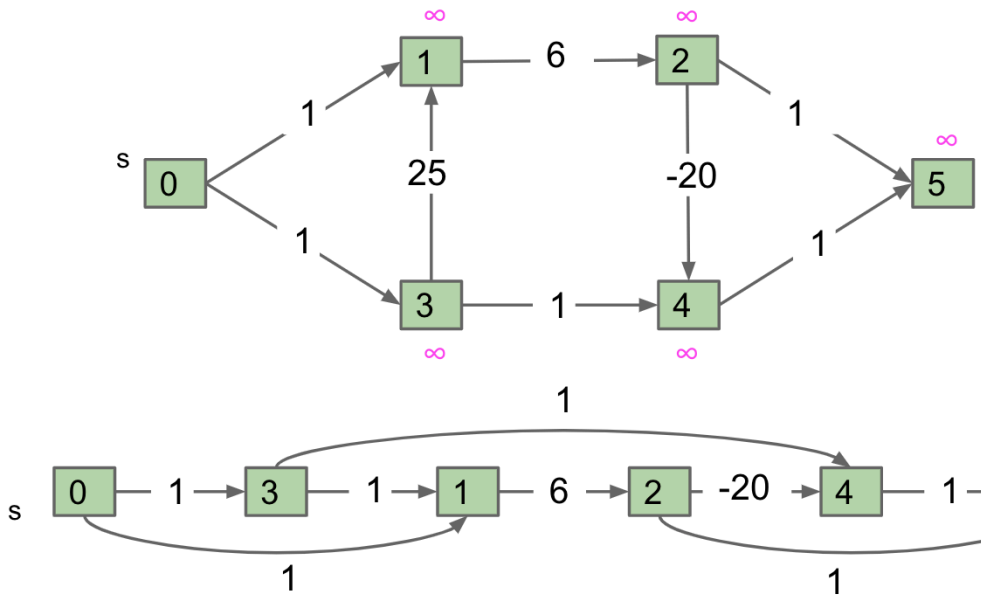
- 从起点开始，把相邻边以及对应点加入heap
- pop最短的边，作为该点的最短路径加入到res中，然后加入该边对应的点的相邻边的积累和以及对应的点
- 直到所有点都被加入到res里
- 类似BFS
- 区别在于：
  - 某点可以多次进入heap，选取最小的pop，pop出来以后，放入res，也就是代表该点已经找到最短路径，不需要再往heap里面加了
  - a点在heap里面可以是不止一次的出现，pop出来的是最短路径
  - 如果res的长度等于N，则全部node的最短路径都已经找到，返回res
- 时间复杂度：
  - O(ElogE), e is length of edges
- 空间复杂度:
  - O(E + N), E is the graph space(领接表) + N is ,ax space used in heap

```
# heap 优化版模型
# 以边为基准
# from lichief

from collections import defaultdict
from heapq import *

# pairs(i, j, weight)   the number of nodes     the start point
def shortestPath(pairs, N, K):
    # transfer pairs into graph [i:{j:w}]
    graph = defaultdict(dict)
    for i, j, w in pairs:
        graph[i][j] = w

    # use the heap to find the shortest path
    # 切记 一定是path在前，因为要以path来排序
    heap = [(0, K)]
    res = {}

    while heap:
        w, i = heappop(heap)

        if i in res:
            continue
        else:
            res[i] = w
```

```
        if len(res) == N:
            return res

        for j in graph[i]:
            if j not in res:
                heappush(heap, (w + graph[i][j], j))

pairs = [[2, 1, 1], [2, 3, 1], [3, 4, 1]]
N = 4
K = 2

shortestPath(pairs, N, K)
```

DAG

First: We have to find a topological order, e.g. 031245. Runtime is O(V + E).



Steps

1. First: We have to find a topological order, Runtime is O(V + E).
2. We have to visit all the vertices in topological order, relaxing all edges as we go. (like Dijkstra) Runtime for step 2 is also O(V + E).

```
# 有待检验
from collections import defaultdict
class Solution:
    def networkDelayTime(self, times: List[List[int]], N: int, K: int) -> int:
        def DFS(i):
            '''
            find the toposort and determine whether there is a circle
```

```
            True: no circle
            False: have circle
            '''
            if visited[i] == 1:
                return True
            if visited[i] == 0:
                return False

            visited[i] = 0
            for out in adj[i].keys():
                if DFS(out) == False:
                    return False

            visited[i] = 1
            topo_sort.append(i)
            return True

        # make adjacent list
        adj = defaultdict(dict)
        for i, j, w in times:
            adj[i][j] = w

        visited = {N:-1 for N in range(1, N + 1)}
        topo_sort = []
        # find the toposort and determine whether there is a circle
        if DFS(K) == False:
            return -1
        if len(topo_sort) != N:
            return -1

        # find the min distance
        topo_sort = topo_sort[::-1]
        dist = {node:float('inf') for node in topo_sort}
        dist[K] = 0

        res = {}
        for i, w in dist.items():
            res[i] = w
            for out in adj[i]:
                dist[out] = min(dist[out], w + adj[i][out])
        return max(res.values())
```

```
public class AcyclicSP {
    private double[] distTo;
```

```
    private DirectedEdge[] edgeTo;

    public AcyclicSP(EdgeWeightedDigraph G, int s) {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];

        //validateVertex(s);

        // 初始化
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        // visit vertices in topological order
        // 拓扑排序顶点，并判断是否有环
        Topological topological = new Topological(G);
        if (!topological.hasOrder())
            throw new IllegalArgumentException("Digraph is not acyclic.");

        // 按照拓扑顺序取出顶点
        for (int v : topological.order()) {
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }

    // relax edge e
    private void relax(DirectedEdge e) {
        int v = e.from(), w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
        }
    }
}
```

3.2.4.3 最长路径

一定要是无环的，因为有环的话，会陷入到无限循环当中

把权重取负数，用最短路径来做

3.2.4.4 最小生成树

不能用Prim's Algorithm和Kruskal's Algorithm的原因：

1->2 8, 1->3 8, 2->3 4, 3->2 3
有平行边的时候，且其它边都是相等的距离，那么不一定能得到最小生成树

3.3 网格搜索

- 易错点：注意grid里面是int还是str
- コツ：
  - 边界条件，可行的条件，i >= 0, i <= len(n) - 1, visited[i] == 0, grid[i] == 1/'1'
  - append之后马上visited[i] = 1，这样不容易错
- 200.Number of Islands
- 733
- 狗家OA，相同岛屿，迷宫死路个数

3.3.1 DFS

```
# 待优化
class MySolution:
    def main(self, grid):
        if not grid:  # corner case
            return None
        if len(grid) == 1:
            return

        # build empty visited with the same size as grid
        visited = [[0 for _ in range(len(grid[0]))] for _ in range(len(grid))]
        # visited =[[0] * len(grid[0]) for _ in range(len(grid))] --> [[0],
[0], [0], [0]]

        # scan grid and dfs the island node
        for i in range(len(grid)):  # len(grid) = 行数
            for j in range(len(gird[i])):  # len(gird[i]) = 每行里面的个数
                if grid[i][j] == 1 and visited[i][j] == 0:
                    self.dfs(i, j, visited, grid)
        return

    def dfs(self, i, j, visited, grid):  # i: 行; j: 列
        if i < 0 or j <0 or i > len(grid) - 1 or j > len(grid[0]) - 1 or
visited[i][j] == 1:  # 结束条件
            return
```

```
        # set visited
        visited[i][j] = 1
        self.dfs(i - 1, j, visited, grid)  # 上走一格
        self.dfs(i + 1, j, visited, grid)  # 下走一格
        self.dfs(i, j - 1, visited, grid)  # 左走一格
        self.dfs(i, j + 1, visited, grid)  # 右走一格
```

3.3.2 BFS

```
# 待优化
class MySolution:
    def main(self, grid):
        if not grid:  # corner case
            return None
        if len(grid) == 1:
            return

        # build empty visited with the same size as grid
        visited = [[0 for _ in range(len(grid[0]))] for _ in range(len(grid))]

        from collections import deque

        for i in range(len(grid)):
            for j in range(len(gird[i])):
                if grid[i][j] == 1 and visited[i][j] == 0:
                    queue.append([i, j])  # 0行0列
                    visited[i][j] = 1
                    while queue:
                        r, c = queue.popleft()  # row 行, cloumn 列
                        if r - 1 >= 0 and visited[r - 1][c] == 0:
                            queue.append([r - 1, c])  # 向上走一格
                            visited[r - 1][c] = 1
                        if r + 1 <= len(grid) - 1 and visited[r + 1][c] == 0:
                            queue.append([r + 1, c])  # 向下走一格
                            visited[r + 1][c] = 1
                        if c - 1 >= 0 and visited[r][c - 1] == 0:
                            queue.append([r, c - 1])  # 向右走一格
                            visited[r][c - 1] = 1
                        if c + 1 <= len(grid[0]) - 1 and visited[r][c + 1] ==
0:
                            queue.append([r, c + 1])  # 向左走一格
                            visited[r][c + 1] = 1
        return
```