Ch3 图

1. 无向图 Undriected Graph

1.1 基本概念

$G=(V,E)$

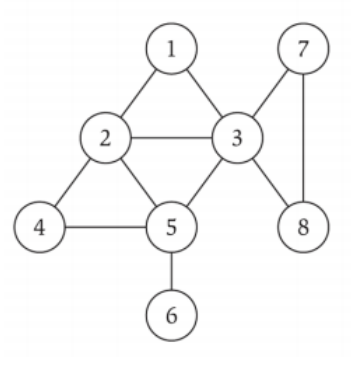
- V：节点的集合
- E：节点之间的边的集合（每条边是一个节点对）

无向图刻画了对象间的两两关系

表示图规模的参数

- 节点个数 $n=|V|$
- 边的条数 $m=|E|$

1.2 无向图示例



如图所示的无向图，有

- $V=\{1,2,3,4,5,6,7,8\}$
- $E=\{1-2,1-3,2-3,2-4,2-5,3-5,3-7,3-8,4-5,5-6\}$
- $n=8$
- $m=11$

1.3 图的应用

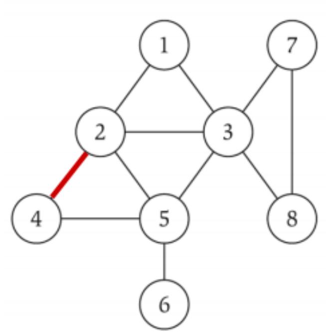
GRAPH	NODES	EDGES
交通网络	街道交口	路
通信网络	计算机	连接它们的线缆
Web	Web 网页	网页的连接关系
社交网络	人	人们的关系
食物链	物种	捕食关系
软件系统	函数	函数调用
任务规划	任务	优先约束
电路	门	电线

1.4 边的表示

邻接矩阵 Adjacency Matrix

一种图的表示方法

$n \times n$ 的矩阵，其中如果 (u, v) 是一条边的话，$A_{uv} = 1$

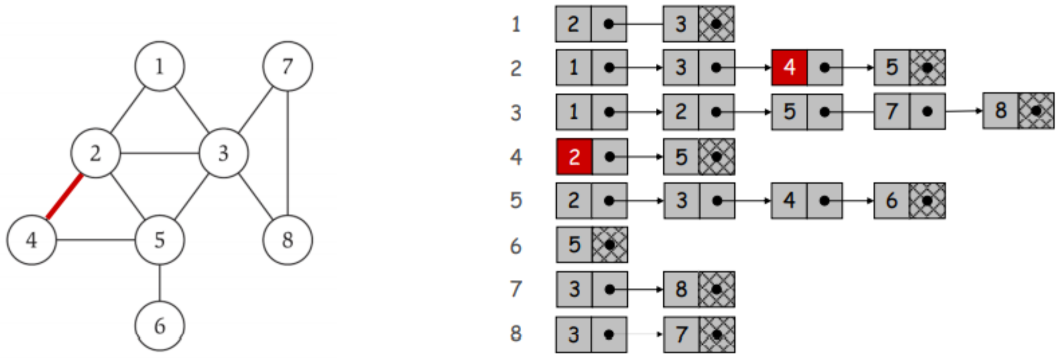


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

- 每条边被表示了 2 次
  - 如 4-2 在邻接矩阵中 [4,2] = 1，[2,4] = 1
- 空间复杂度正比于 n^2
- 检查 (u, v) 是否为一条边的时间复杂度为 $\Theta(1)$
- 确认所有边需要的时间复杂度为 $\Theta(n^2)$

邻接链表 Adjacency List

节点索引的列表数组



- 每条边会被表示两次
- 空间复杂度为 $m+n$
- 检查 (u,v) 是否为一条边的时间复杂度为 $O(deg(u))$
    - $degree$ = 节点 u 的出度
- 确认所有边需要的时间复杂度为 $\Theta(m+n)$

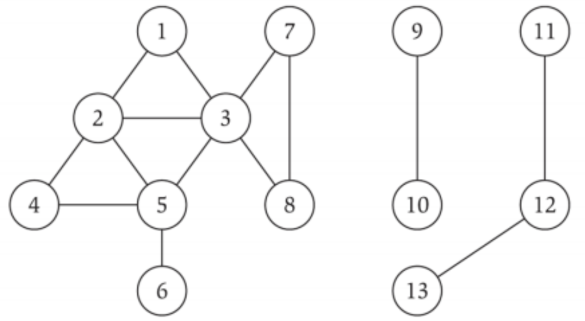
1.5 图的定义

路径 Path

- 无向图 $G=(V,E)$ 中的一条路径是由节点 $v_1, v_2, \ldots, v_{k-1}, v_k$ 组成的序列 P，其中每个连续对 (v_i, v_{i+1}) 是在 E 中的一条边
- 如果所有节点都是不同的，那么路径被称为简单(simple)的

连通性 Connectivity

- 如果对于每一对节点 u 和 v，u 和 v 之间都有一条路径，那么我们说这个无向图是连通的



环 Cycle

- 一个循环是路径 $v_1, v_2, ..., v_{k-1}, v_k$，其中 $v_1 = v_k$ $(k > 2)$，并且前 $k-1$ 个节点都是不同的

树 Tree

如果无向图是连通的且不包含环，则它是树

对于一个有 n 个节点的无向图 G，下面任何两个条件满足都能推导出第三个

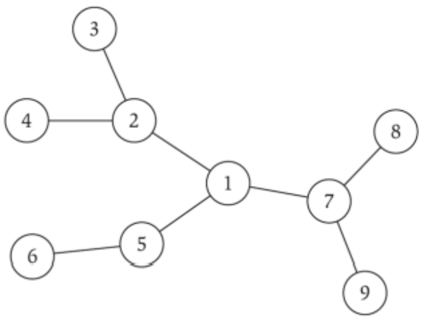
- G 是连通的
- G 没有环
- G 有 $n-1$ 条边

即

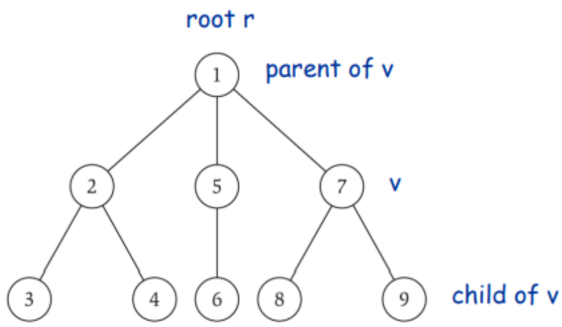
- 这三条中，满足任意两条可以作为的图为树
- 树满足以上三个定义

有根树 Rooted Tree

给定一棵树 T，选择一个根节点 r 向外展开各条边，并将每条边朝向远离 r 的方向



a tree

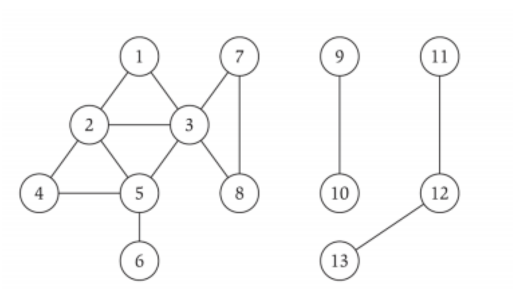


the same tree, rooted at 1

每个有 n 个节点树都有 $n-1$ 条边

连通分支 Connected Component

s 的连通分支指从 s 可达的所有节点的集合



- 如图，节点 1 的连通分支为 $\{1,2,3,4,5,6,7,8\}$

2. 无向图的遍历 BFS

2.1 问题描述

连通性问题

给定两个节点 s 和 t，判断在它们中间是否存在一条路径

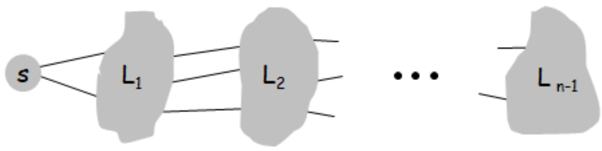
最短路径问题

给定两个节点 s 和 t，它们之间最短路径的长度是多少

2.2 算法

从 s 向外探索所有可能的方向，每次添加一个“层”的节点，这些节点不在之前已经遍历到的节点中

算法执行流程



- $L_0 = \{s\}$
- $L_1 =$ 所有 L_0 的节点的邻居
- $L2 =$ 所有不属于 L_0 或 L_1，但与 L_1 中的节点有边相连的节点
- $L_{i+1} =$ 所有不属于之前的层的节点，并且在 L_i 中有一个节点的边的节点

BFS 伪代码

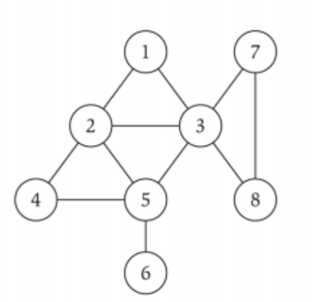
```
// G BFS对应的图
// s 初始节点ha
BFS(G,s){
    initial all node;// not vivsited yet
    create hash-set S;// maitain the node that have visited
    create queue Q;
    S.add(node); // that has been visited
    Q.enqueue(s);

    while(Q is not null){
        node = Q.dequeue();
        outnodes = all node that is conntect to node;
        for outnode in outnodes{
            if(S is not contain(outnode)){
                Q.enqueue(node);
                S.add(node)
            }
        }
    }
}
```

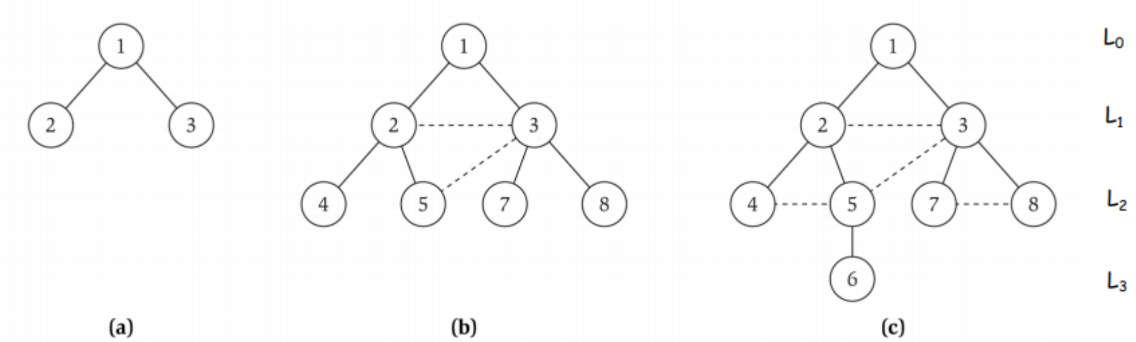
2.3 定理

对于每一个 i，L_i 包含了所有与s 的距离正好为 i 的节点（没有边权），存在从 s 到 t 的路径，当且仅当 t 属于某一层

2.4 性质

令 T 为无向图 $G=(V,E)$ 的一颗 BFS 树 (x,y) 为 G 的一条边，则 x 和 y 在 T 中的层数至多相差 1





2.5 时间复杂度分析

如果图是由邻接链表给出的，那么 BFS 的时间复杂度是 $O(m+n)$

证明

- 当我们考虑一个节点 u 的时候，有 $deg(u)$ 条边
- 我们会遍历 n 个节点
- 总处理边的时间为 $\sum_{u\in V} deg(u)=2m$，每条边既在 u 的邻接链表里出现，也在 v 的邻接链表里出现
- 故整体时间复杂度为 $O(m+n)$

3. 无向图的遍历 DFS

3.1 问题描述

给定两个节点 s 和 t，判断在它们中间是否存在一条路径

3.2 算法

从 s 向外探索进行深度优先搜索

1. 创建栈 S，将 s 放入 S 内 $\{s\}$
2. 当栈不为空的时候
    a. 栈弹出节点 v
    b. 对于 v 的所有边的节点
        i. 如果有节点没有访问，则将其压栈
        ii. 如果所有节点已经访问了，弹出 v

3.3 时间复杂度分析

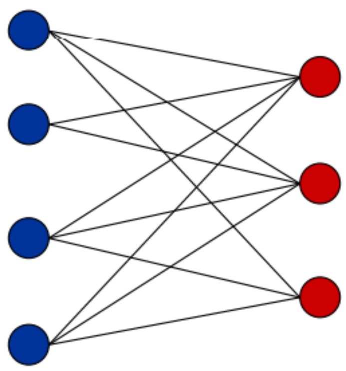
DFS 的时间复杂度为 $O(m+n)$

4. 二部图检测 Testing Bipartiteness

3.1 概念定义

二部图

一个无向图 $G=(V,E)$ 是二部图，如果其节点可以被标记成红的或蓝的，使得每条边都有一个红端和一个蓝端



a bipartite graph

奇圈

图中有奇数个节点的环

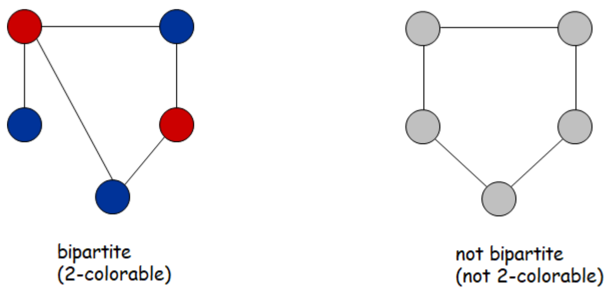
二部图不包含奇圈

如果一个图 G 是二部图，那么它不可能包含有奇圈

证明

- 如果 G 中包含奇圈，在不可能对它用两种颜色交替着色

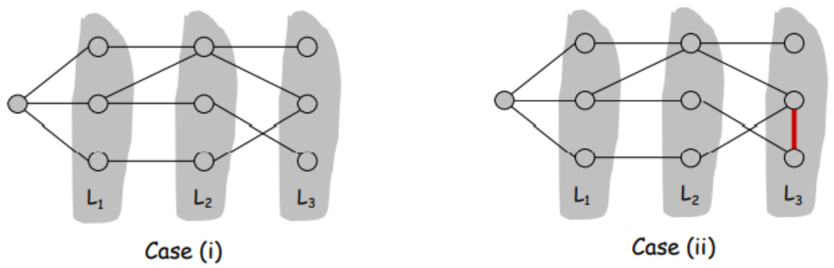
如下图所示



- 左图不包含奇圈，是二部图
- 右图包含奇圈，不能着色

BFS 在二部图的性质

设 G 是连通图，设 $L_0, \ldots, L_k$ 是 BFS 从节点 s 起始点产生的层



- 如果 G 没有边连接同一层的两个节点，G 是二部图（Case 1）
- 如果 G 的一条边连接同一层的两个节点，G 包含一个含有奇数个节点的环，也就是说它不是一个二部图（Case 2）

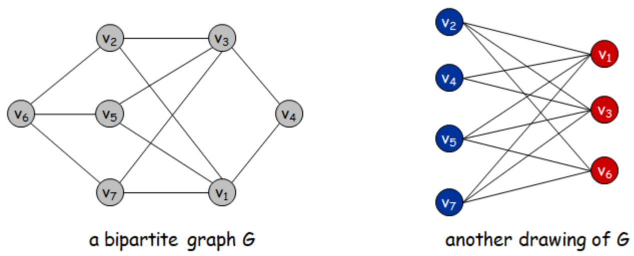
二部图的判定

一个图 G 是二部图，当且仅当它不包含奇圈

如果一个图 G 不包含奇数个节点的环，它就是二部图

3.2 问题描述

给定一个图 G，判断它是否是一个二部图



3.3 算法

使用 BFS + 染色标记法

```
// G 判断二部图所对应的节点
// s 初始节点
Judgement(G,s){
    create hash-set Red;// color 1
    create hash-set Blue;// color 2
```

```
    create queue Q;

    Q.enqueue(s);
    Red.add(s);
    while(Q is not null){
        node = Q.dequeue();
        outnodes = all node that is conntect to node;
        for outnode in outnodes{
            if(Both Blue and Red not contain outnode){
                node in Blue?Red.add(outnode):Blue.add(outnode);//
                Q.enqueue(outnode);
            }else if(outnode in the different hash-set as node){
                continue;
            }else if(outnode in the same hash-set as node){
                return false;
            }
        }
    }
    return true;
}
```

3.4 时间复杂度分析

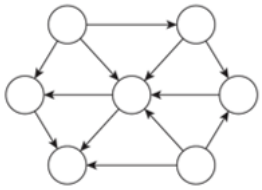
由于使用 BFS，时间复杂度为 $O(m+n)$

3.4 二部图的应用

- 在婚姻匹配问题中，红色代表男性，蓝色代表女性
- 在调度问题中，红色代表机器，蓝色代表工作
- 独立集问题适合使用二部图

5. 有向图 Driected Graph

5.1 基本概念



$G = (V, E)$

- V = 节点的集合
- E = 节点之间的边的集合

边是有方向的，从节点 u 指向 v

表示图规模的参数

- 节点个数 $n = |V|$
- 边的条数 $m = |E|$

5.2 关于有向图的问题

有向图的连通性问题

给定一个节点 s，找到所有从 s 出发，可达的节点

有向图的最短路径问题

给定两个节点 s 和 t，它们之间最短路径的长度是多少

图搜索问题

BFS 可以自动拓展到有向图上

5.3 有向图的应用

- 网络爬虫：从网页 s 开始，找到所有与 s 直接或间接链接的网页

6. 图的强连通性 Strongly Connected

6.1 概念定义

相互可达

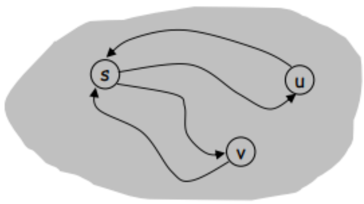
如果 u 到 v 有一条路径， v 到 u 也有一条路径，则节点 u 和 v 是相互可达的

强连通

一个图是强联通的，如果每一对节点是相互可达的

强连通的充分必要条件

设 s 为任意节点，G 是强连通的，当且仅当从 s 出发，每个节点都可以到达，并且从每一个节点出发，s 是可达的



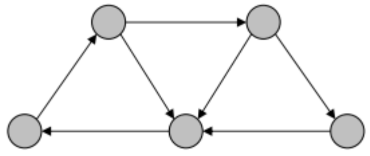
- 上图，节点 s，u，v 相互可达，故这个图是强连通的

6.2 问题描述

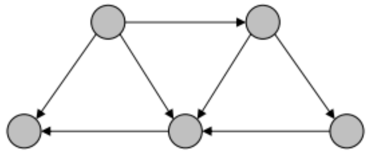
给定一个图 G，判断它是否是强连通的

6.3 算法

BFS 判断强连通性



strongly connected



not strongly connected

1. 随便挑出一个节点 s
2. 对 s 在 G 上进行 BFS 算法
3. 对 s 在 G^{rev} 上进行 BFS （其中 G^{rev} 是将原来的 G 中的每一条边反向）
4. 如果在两次 BFS 运行中所有的节点都能访问到，那么图是强连通的

6.4 时间复杂度分析

通过 BFS 判断图是否有强连通性，时间复杂度为 $O(m+n)$

7. 拓扑排序

7.1 概念定义

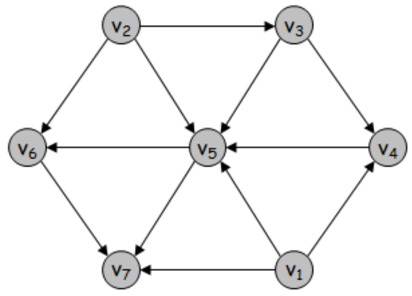
有向无环图定义 DAG

DAG 是一个不包含有向环的有向图

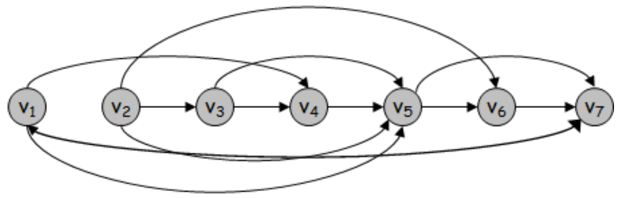
优先约束：$edge(v_i, v_j)$意味着 v_i 必须在 v_j 之前

拓扑排序 Topological Order

有向图 $G=(V,E)$ 的拓扑顺序是其节点 v_1, v_2， ...， v_n 的顺序，使得对于每条边 (v_i, v_j) 都有 $i<j$



a DAG



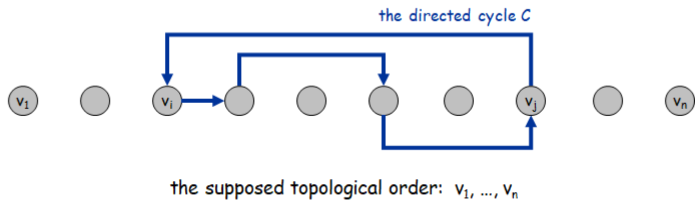
a topological ordering

- 上图为一个有向图和它的拓扑排序

拓扑排序和 DAG

如果 G 有一个拓扑顺序，那么 G 是一个 DAG

- 假设拓扑排序可以有有向环
- 设 G 有一个拓扑排序 $v_1, \ldots, v_n$，并且 G 有一个有向环 C



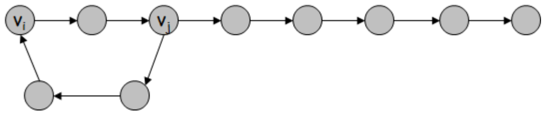
- 令 v_i 为 C 中标号最小的节点，v_j 为 C 中在 v_i 前的节点，因此 (v_j, v_i) 是一个边
- 根据我们之前的选择 $i < j$
- 因为 (v_j, v_i) 是一条边，$v_i, \ldots, v_j$ 是一个拓扑排序，所以一定有 $j < i$（矛盾）
- 故得证

如果 G 是一个 DAG，那么 G 有一个拓扑排序

- 当 $n = 1$ 时，显然正确
- 假设一个 DAG 有 $n - 1$ 个节点，且它有一个拓扑排序
- 给定含 n 个节点的 DAG，则根据 DAG 没有入边的节点（见下方证明），可知存在节点 v 没有入边
- 因为删去 v 不会产生入边，所以 $G - \{v\}$ 仍然是一个 DAG，根据假设，在 DAG 有 $n - 1$ 个节点时，它有拓扑排序
- 故归纳法得证

没有入边的节点

如果 G 是一个 DAG，那么 G 一定存在没有入边的节点

- 设 G 是一个有向无环图，并且 G 的每一个节点都至少有一条入边


- 设 G 有一个拓扑排序 $v_1, \ldots, v_n$
- 令 v_i 为拓扑排序中标号最小的节点
- 因为 v_i 有一条入边，令v_j 为拓扑排序中在 v_i 后的节点，即 $j > i$
- 因为 (v_j, v_i) 是一条边，且$v_i, \ldots, v_j$ 是一个拓扑排序，所以一定有 $j < i$（矛盾）
- 故得证

7.2 问题描述

判断一个图是否有拓扑排序 / 是否为 DAG

7.3 算法

1. 找到一个入度为 0 的节点 v，并将其放在第一个
2. 把 v 和与其相连的边从 G 中删掉
3. 递归计算 $G - \{v\}$ 的拓扑顺序，并在 v 后面加上这个顺序

维护下列信息

- $count[w]$ = 剩余的入边数
- S = 没有入边的剩余节点的集合

初始化

- $O(m + n)$ 单次扫描图

更新

- 从 S 中移除 v
- 从 v 到 w 的所有边的递减计数 $count[w]$，如果计数 $count[w]$ 达到 0，将 w 添加到 S
- 每条边是 $O(1)$

伪代码如下

```
Topological-Order(G){
    create queue Q;
    Q.enqueue(Nodes that the in-degree = 0);
    count = in-degree = 0 node number;
    N = the number of nodes in G
    create list L;// record the topologiacl order

    while(Q is not null){
        node = Q.dequeue();
        L.add(node)
        for outnode in outnodes{
            delete the edge between outnode and node;
            if(outnode in-degree = 0){
                Q.enqueue(outnode);
                count++;
            }
        }
    }

    if(count = N){
        print(L);
    }else{
        return false;// there is not topological order
    }
}
```

7.4 时间复杂度分析

算法在 $O(m+n)$ 时间内找到一个拓扑顺序