图论基础

深圳大学

关注乃琳Quee谢谢喵

图的概念

Concept of Graph

模型引入

- 你常去深大的哪些地点?
- 把每个地点抽象为平面上的点，再把每条道路抽象为平面上的线。
- 这个时候就形成了一张图。

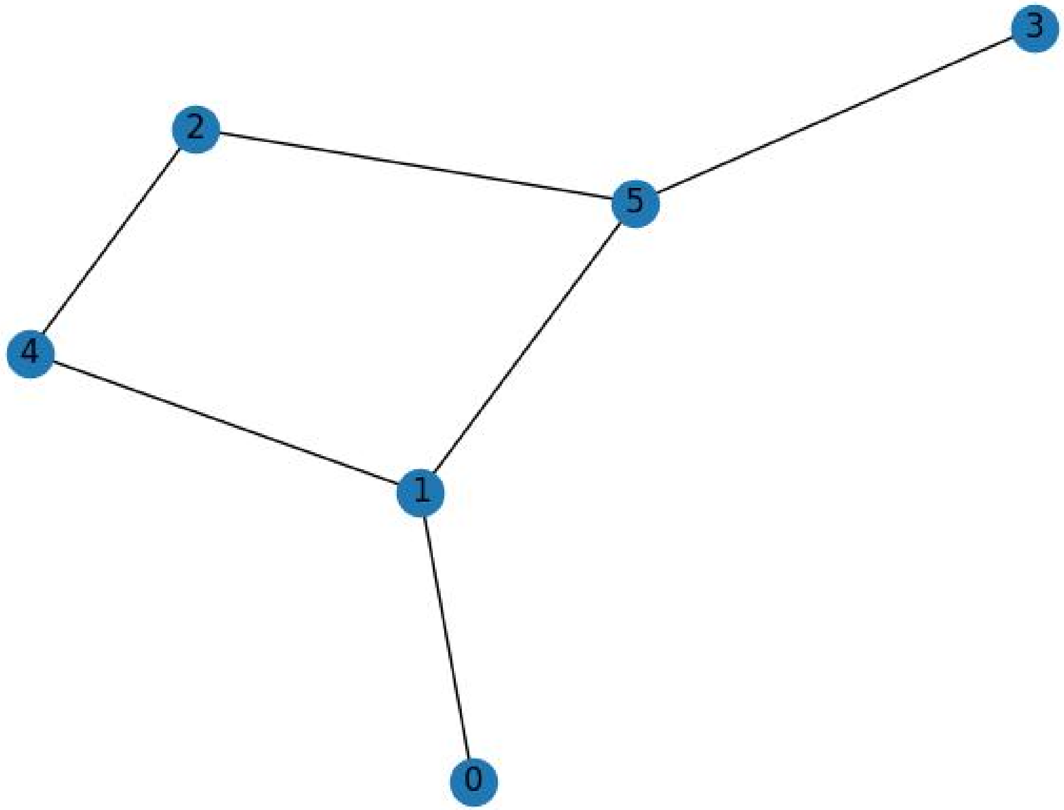
图的定义

- 图形是离散数学的研究对象之一。

- 二元组的定义

图G是一个有序二元组(V,E)，其中V称为顶集(Vertices Set)，E称为边集(Edges set)，E与V不相交。它们亦可写成V(G)和E(G)。其中，顶集的元素被称为顶点(Vertex)，边集的元素被称为边(edge)。

举个例子



顶集 V = {0, 1, 2, 3, 4, 5}
边集 E = {(0, 1), (1, 4), (2, 4), (2, 5), (3, 5), (1, 5)}

有向图与无向图

- 图可以分为有向图、无向图和混合图。

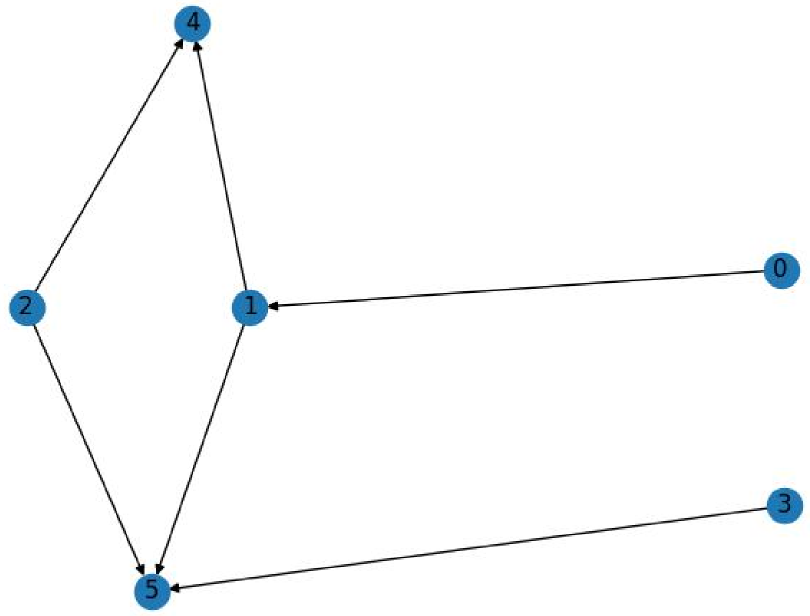
- 如果给图的每条边规定一个方向，那么得到的图称为有向图。在有向图中，与一个结点相关联的边有出边和入边之分。相反，边没有方向的图称为无向图。

- 一个图既有无向边又有有向边就可以称为混合图。
- 虽然你也可以把所有图理解为有向图，即一条无向边可以拆为两条正反向的有向边。

理解有向边

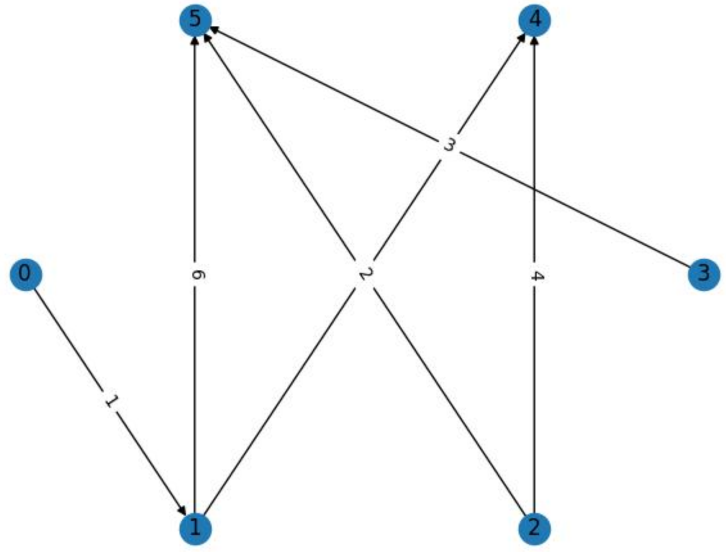
- 可以看成现实生活中的单行道，人为规定只允许单向通过。

- 举个例子：



赋权图

- 边集 E 中元素由二元组(u, v)变为三元组(u, v, c)，表示一条边所连接的两个顶点与这条边的边权值。
- 赋权图中每一条边都有一个权值。
- 若图中权值均为正实数，称其为正权图。



度数

- 在无向图中，一个点的度数就是它所连接的边的数量，记作$d(u)$.
- 在有向图中，度数分为入度$d^-(u)$和出度$d^+(u)$，分别表示指向该点和指出该点的边的数量。

简单图

- 自环：对于 E 中的一条边(u, v)，若 u = v 则称其为自环。
- 重边：若 E 中存在两条边e1, e2，它们完全相同，则称其为重边。

- 简单图：若一个图中没有自环和重边，它被称为简单图。

> **Warning**
>
> 在无向图中 (u, v) 和 (v, u) 算一组重边，而在有向图中，$u \to v$ 和 $v \to u$ 不为重边。

> **Warning**
>
> 在题目中，如果没有特殊说明，是可以存在自环和重边的，在做题时需特殊考虑。

途径

- 赶往上课的途中，你是否会按照基本固定的路线行走？

- 途径是连接一连串顶点的边的序列，可以为有限或无限长度。

- 形式化地说：一条有限途径ω是一个边序列$e_1, e_2, e_3, \ldots, e_k$，满足存在一个顶点序列$v_0, v_1, v_2, \ldots, v_k$有$e_i = (v_{i-1}, v_i)$。途径也可简写为$v_0 \to v_1 \to v_2 \to \ldots \to v_k$。
- 通常来说，边的数量 k 被称作这条途径的长度（如果边是带权的，长度通常指途径上的边权之和，题目中也可能另有定义）。

途径的不同形态

- 迹：对于一条途径ω，若$e_1, e_2, e_3, \dots, e_k$两两互不相同，则称$\omega$是一条迹。

- 简单路径：对于一条迹ω，若其连接的点的序列中点两两不同，则称ω是一条简单路径。

- 回路：对于一条迹ω，若$v_0 = v_k$，则称ω是一条回路。
- 环：对于一条回路ω，若$v_0 = v_k$是点序列中唯一重复出现的点对，则称ω是一个环。

连通（无向图）

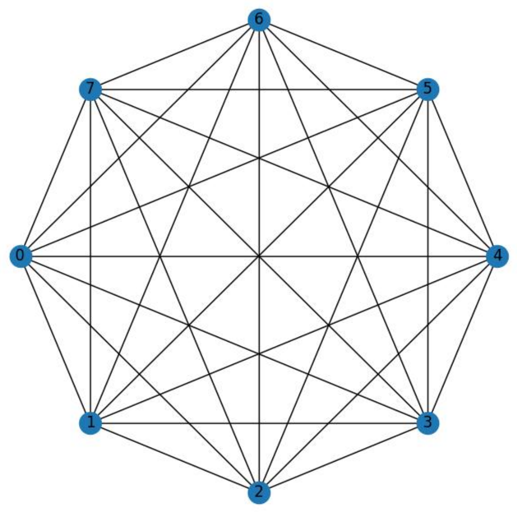
- 若点u与点v之间存在一条路径可以互相到达，则称两点连通。
- 连通图：图中任意两点之间连通。

连通（有向图）

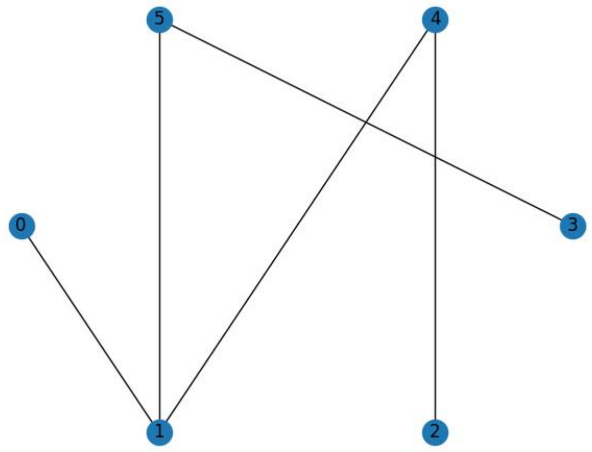
- 若点u存在一条路径可达点v，则称u可达v。
- 强连通：图中任意两点之间互相可达。
- 弱连通：本来非强连通的图，单向边全部换为双向边后强连通。

稀疏图/稠密图

- 若一张图的边数远小于其点数的平方，那么它是一张稀疏图。
- 若一张图的边数接近其点数的平方，那么它是一张稠密图。



稠密图



稀疏图

图的存储与遍历

Storage and Traversal of Graphs

遍历

- 所谓遍历(Traversal)，是指沿着某条搜索路线，依次对图中每个顶点均做一次访问。

从定义出发

- 直接存储图 G 的顶集 V 和 边集 E（约定n = |V|， m = |E|）

```
int n, m;
int V[N];
struct Edge{ int u, v; }E[M];
```

- 查询点、边的存在性：O(n)、O(m)
- 查询一个点的所有出边：O(m)
- 遍历一个图O(nm)

从定义出发的存储方式的遍历

- 需要使用dfs进行遍历。

- 由于复杂度大，代码量大，难写难用，OI/ACM中不会用到这种存储方式的遍历，所以此处不予展示。

邻接矩阵

- 使用一个二维数组 a[i][j] 为0或1表示i -> j这条边是否存在。
- 如果是赋权图，a[i][j]可以直接存储i -> j的边权。
- 以无向赋权图为例：

查询边的存在性：O(1)
遍历一个图：O(n^2)

```cpp
int adj[N][N];
void solve(){
    int n, m; cin >> n >> m;
    for(int i = 1; i <= m; i++){
        int u, v, w; cin >> u >> v >> w;
        adj[u][v] = w;
        adj[v][u] = w;
    }
}
```

- 由于邻接矩阵在稀疏图上效率很低（尤其是在点数较多的图上，空间无法承受），所以一般只会在稠密图上使用邻接矩阵。

邻接矩阵的遍历

```
for(int u = 1; u <= n; u++){
        for(int v = 1; v <= n; v++){
            cout << " u -> v " << adj[u][v] << endl;
        }
    }
```

邻接表

- 使用一个支持动态增加元素的数据结构构成的数组，如 vector<int> a[n + 1] 来存边，其中 a[u] 存储的是点u的所有出边的相关信息（终点、边权等）。
- 以无向赋权图为例：

```cpp
vector<pair<int,int>> adj[N];
void solve(){
    int n, m; cin >> n >> m;
    for(int i = 1;i <= m; i++){
        int u, v, w; cin >> u >> v >> w;
        adj[u].push_back(make_pair(v, w));
        adj[v].push_back(make_pair(u, w));
    }
}
```

查询是否存在u -> v，复杂度O(|d(u)|)

离线后可以变为O(log|d(u)|)

遍历一张图O(n+m)

邻接表的遍历

```cpp
for(int u = 1; u <= n; u++){
    for(int i = 0; i < adj[u].size(); i++){
        cout << u << " -> " << adj[u][i].first << " w = " << adj[u][i].second << endl;
    }
}
```

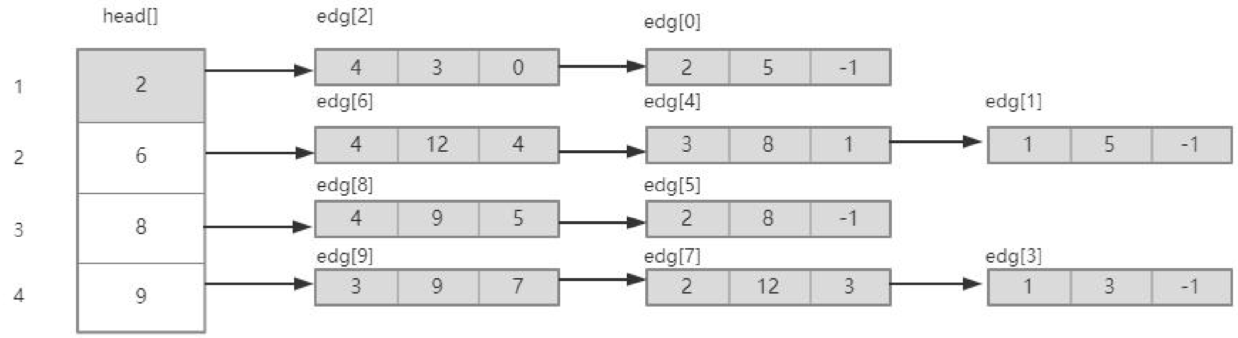
个人比较喜欢的写法

```cpp
void solve(){
    int n, m; cin >> n >> m;
    vector<vector<array<int,2>>> adj(n + 1);
    for(int i = 1; i <= m; i++){
        int u, v, w; cin >> u >> v >> w;
        adj[u].push_back({v, w});
        adj[v].push_back({u, w});
    }
    for(int i = 1; i <= n; i++){
        for(auto [v, w] : adj[i]){
            cout << i << " -> " << v << " w = " << w << endl;
        }
    }
}
```

链式前向星

- 可以理解为链表实现的邻接表。
- 因为好写且常数小所以经常在OI/ACM中使用。

```
struct edge{
    int to,w,next;
}edge[N];
int head[N],cnt;
void init(){
    memset(head,-1,sizeof(head));
    cnt=0;
}
void add_edge(int u,int v,int w){
    edge[cnt].to= v;
    edge[cnt].w=w;//权值
    edge[cnt].next=head[u];
    head[u]=cnt++;
}
```



查询是否存在u -> v，复杂度O(|d(u)|)

遍历一张图O(n+m)

链式前向星的遍历

```
void solve(){
    init();
    int n, m; cin >> n >> m;
    for(int i = 1; i <= m; i++){
        int u, v, w; cin >> u >> v >> w;
        add_edge(u, v, w);
        add_edge(v, u, w);
    }

    for(int u = 1; u <= n; u++){
        for(int i = head[u]; ~i; i = edge[i].next){
            int v = edge[i].to, w = edge[i].w;
            cout << u << " -> " << v << " w = " << w << endl;
        }
    }
}
```

以DFS的方式遍历图

- 以链式前向星为例（包括BFS）

```cpp
int vis[N];
void dfs(int u){
    vis[u] = 1;
    cout << u << " visited!" << endl;
    for(int i = head[u]; ~i; i = edge[i].next){
        int v = edge[i].to, w = edge[i].w;
        if(vis[v]) continue;
        dfs(v);
    }
}
```

以BFS的方式遍历图

```cpp
int vis[N];
void bfs(int st){
    memset(vis, 0, sizeof(vis));
    queue<int> que;
    que.push(st);
    vis[st] = 1;
    while(!que.empty()){
        int u = que.front(); que.pop();
        cout << u << " visited!" << endl;
        for(int i = 0; i < adj[u].size(); i++){
            int v = adj[u][i].first, w = adj[u][i].second;
            if(vis[v]) continue;
            vis[v] = 1;
            que.push(v);
        }
    }
}
```

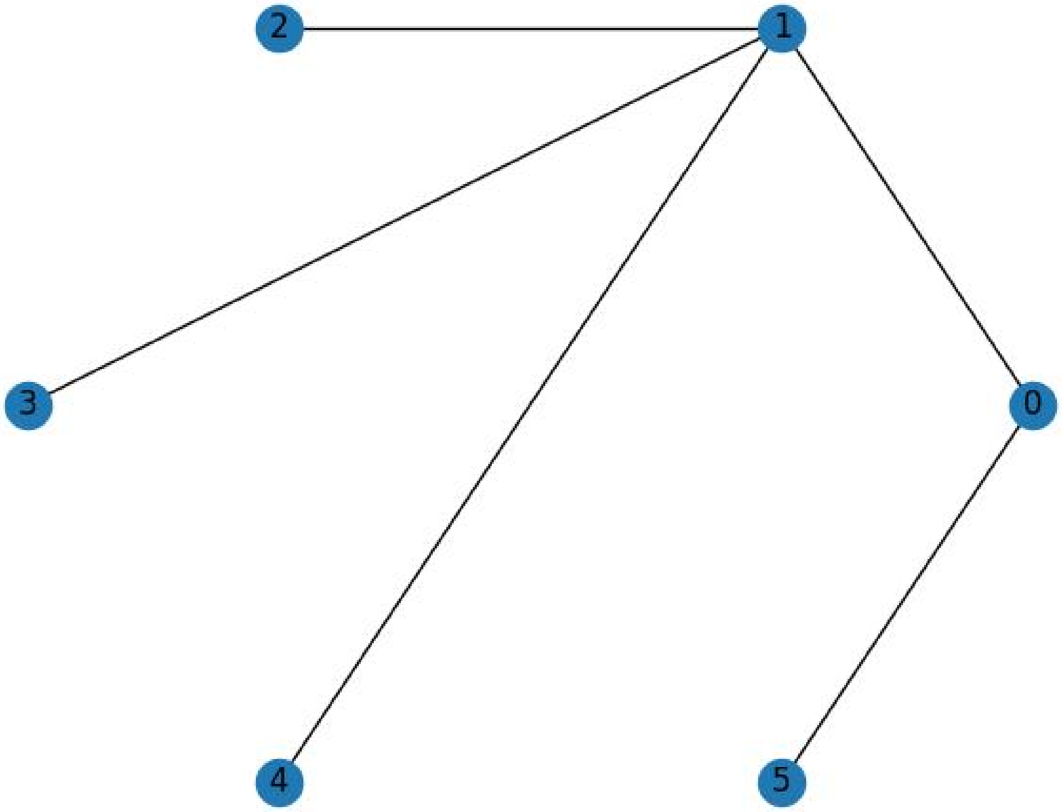
树的概念

Concept of Tree

模型引入

- 你是否有观察过树?
- 树是一个n个点，n - 1 条边的无向连通图。
- 任意两个结点之间有且仅有一条简单路径的无向图。

举个例子



有根树

- 对于一棵树而言，我们指定它其中的一个结点为根，那么这棵树就被称为有根树。
- 对于有根树而言，结点之间就有了上下级关系。

一些适用于无根树和有根树的定义

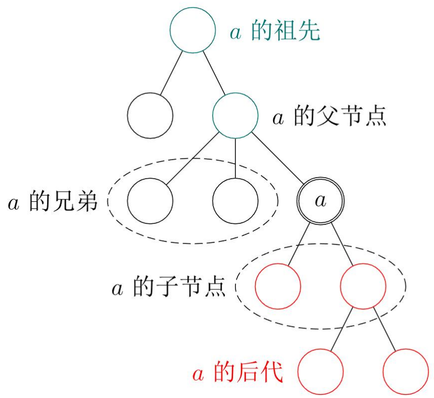
- 森林：若干棵互不连通的树构成的图

- 无根树的叶子结点：度数不超过1的结点
  (n=1时存在度为0的叶结点)
- 有根树的叶子结点：没有子结点的结点

- 树的直径：树上任意两节点之间最长的简单路径。

一些关于有根树的定义

- 父结点：对于除根以外的每个结点，定义为从该结点到根路径上的第二个结点。 根结点没有父结点。
- 祖先：一个结点到根结点的路径上，除了它本身外的结点。 根结点的祖先集合为空。
- 子结点：与u相连的、除了父结点的都被称为子结点
- 兄弟：同一个父亲的多个子结点互为兄弟。
- 后代：子结点和子结点的后代。 （与祖先相反）

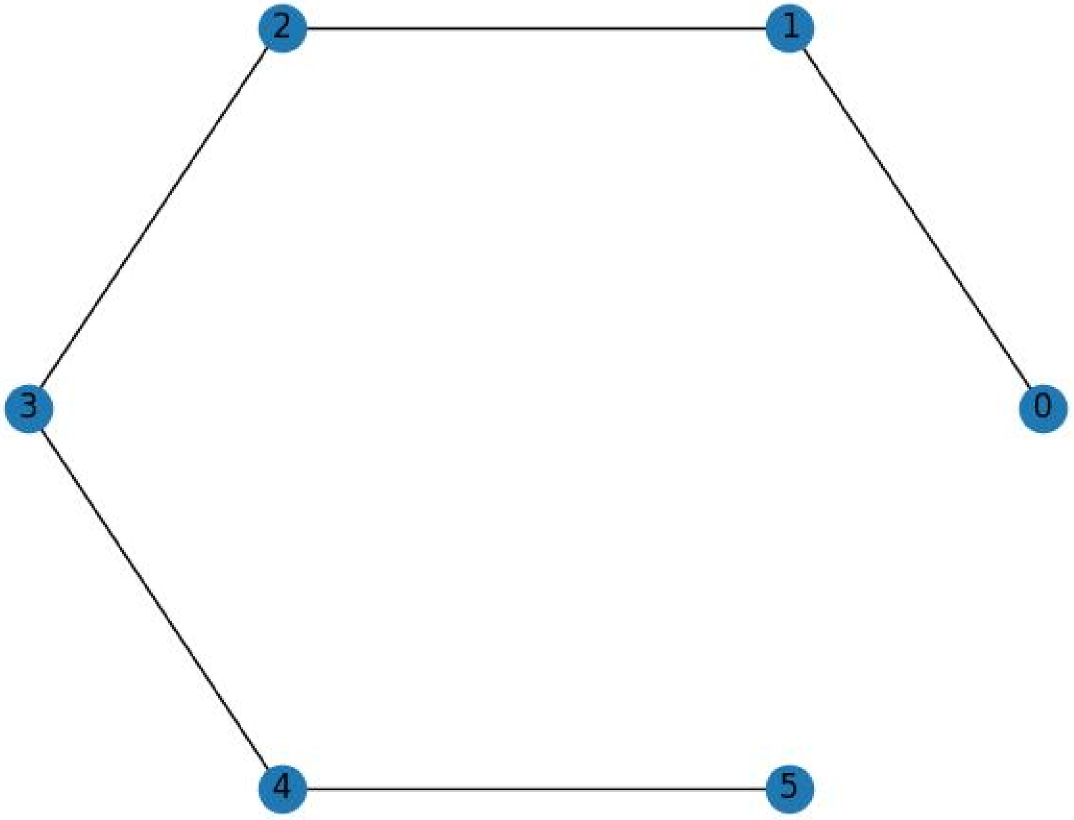


一些关于有根树的定义

- 子树：删掉与父亲相连的边后，该结点所在的子图。（其实就是一棵以当前结点为根的树）
- 结点的深度：到根结点的路径上的边数。
- 树的高度：所有结点的深度的最大值。

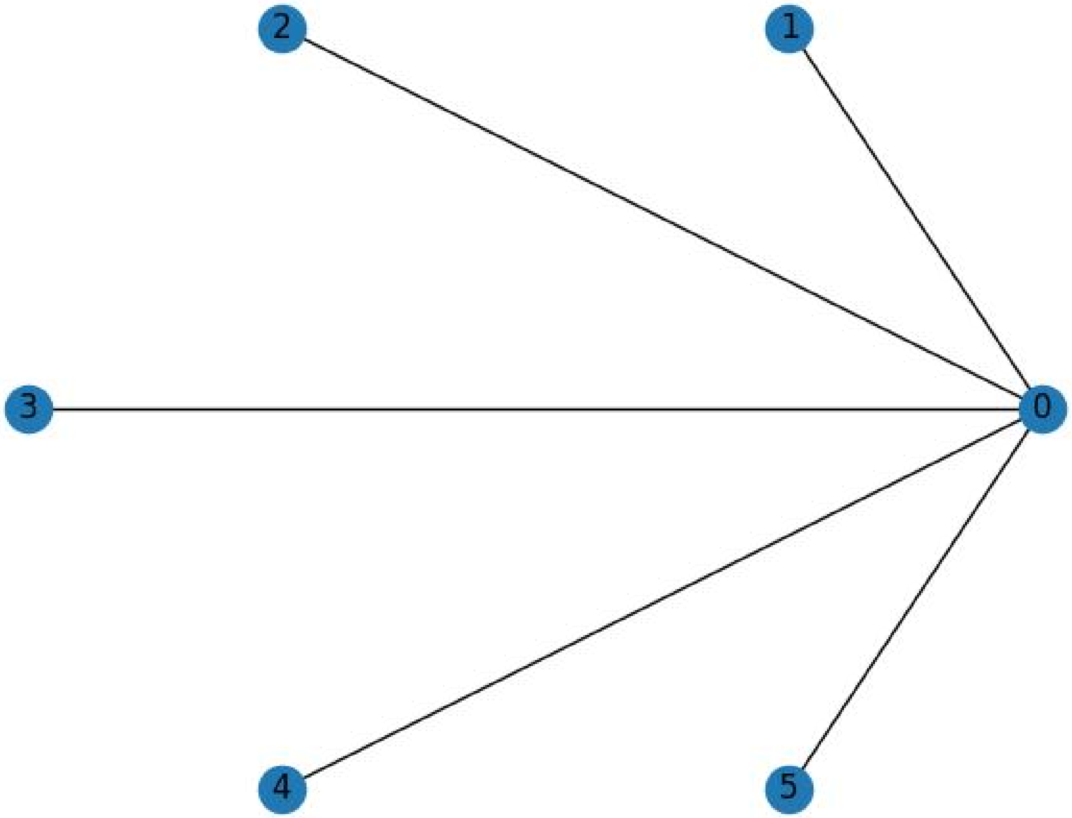
特殊的树的形态

- 链



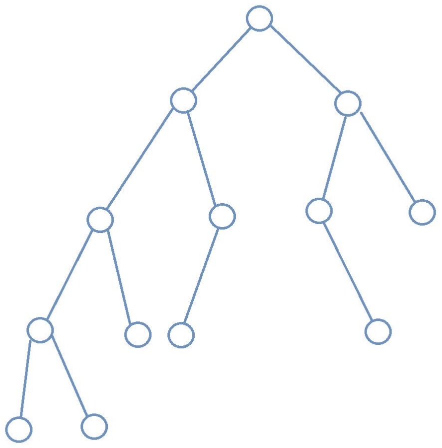
特殊的树的形态

- 菊花图



特殊的树的形态

- 二叉树
- 每个结点最多只有两个儿子（子结点）的有根树称为二叉树。常常对两个子结点的顺序加以区分，分别称之为左子结点和右子结点。



树的存储与遍历

Storage and Traversal of Trees

邻接表存储与链式前向星存储

- 和图的存储一致
- 推荐这两种（对于任何类型的树都可以使用）

DFS遍历

```
void dfs(int u, int f){
    dep[u] = dep[f] + 1; // 用dep数组记录每一个节点的深度
    fa[u] = f;           // 记录每一个点的父亲节点
    for(int i = 0; i < adj[u]; i++){
        int v = adj[u][i].first, w = adj[u][i].second;
        if(v == f) continue;
        dfs(v, u);
    }
}
```

二叉树的特殊存储

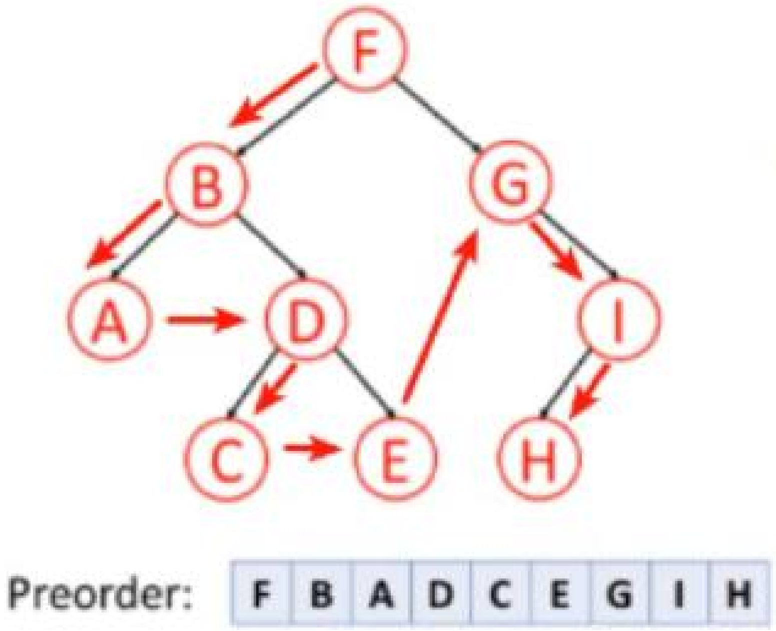
- 由于二叉树具有子结点少的特性，对于其存储方式往往较为特殊。
- 在题目给定左右儿子时一般采用左右儿子记录法（有根树）。
- 否则使用和普通树一致的写法可能会更好使用一些

```
struct Node{
    int ls, rs;
    Node(){ ls = 0, rs = 0; }
    Node(int l, int r){ ls = l, rs = r; }
}a[N];

int main(){
    cin>>n>>root;
    for(int i = 1; i<=n; ++i){
        cin>>ls>>rs;
        a[i] = Node(ls, rs);
    }
}
```

二叉树遍历——先序遍历

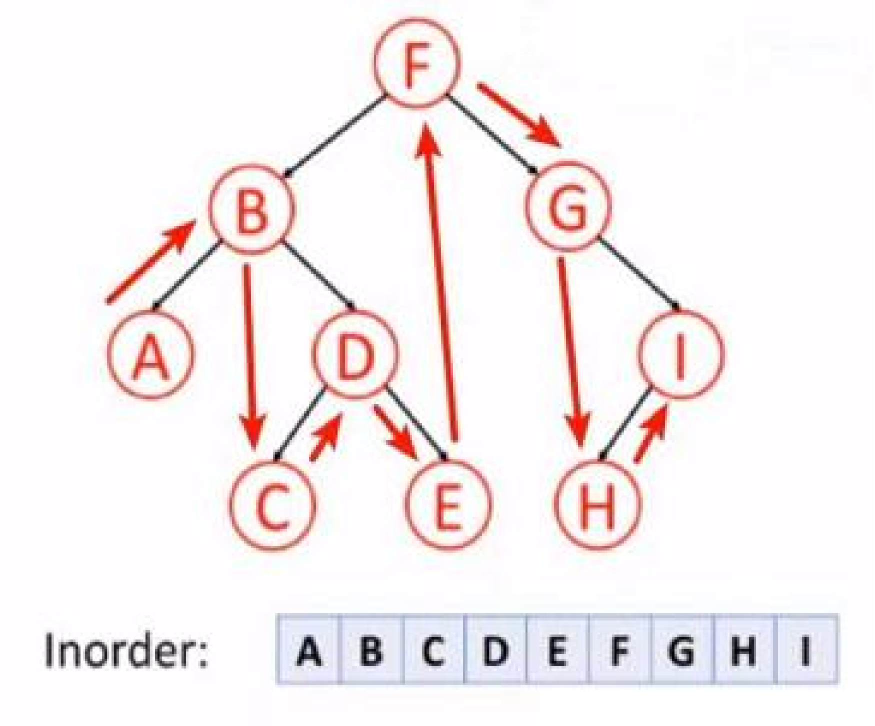
- 按照 **根**，**左**，**右** 的顺序遍历二叉树。




```c
void pre(int x){
	if(!x) return;
	printf("%d ", x);
	pre(a[x].ls);
	pre(a[x].rs);
}
```

二叉树遍历——中序遍历

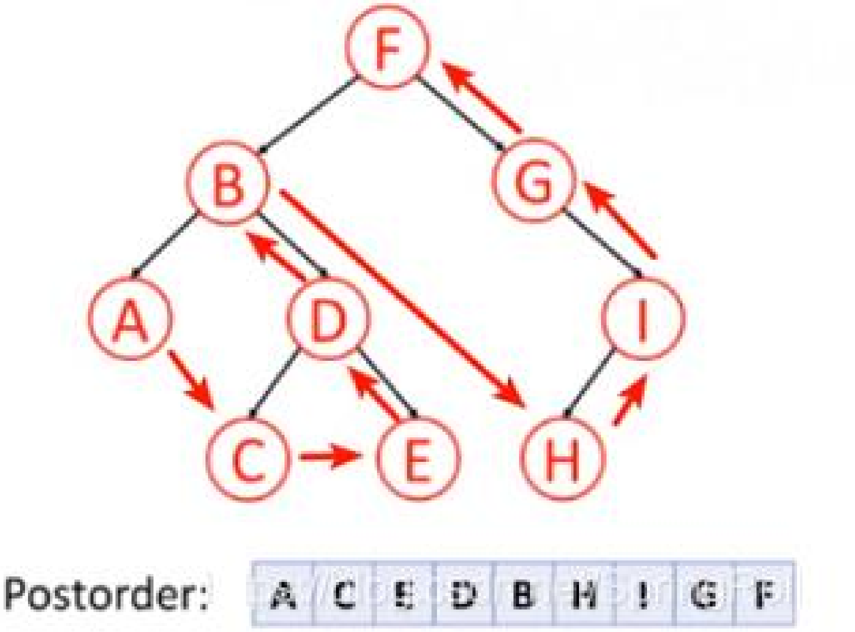
- 按照 **左**，**根**，**右** 的顺序遍历二叉树。




```
void mid(int x){
    if(!x) return;
    pre(a[x].ls);
    printf("%d ", x);
    pre(a[x].rs);
}
```

二叉树遍历——后序遍历

- 按照 **左**，**右**，**根** 的顺序遍历二叉树。




```
void last(int x){
    if(!x) return;
    pre(a[x].ls);
    pre(a[x].rs);
    printf("%d ", x);
}
```

反推

- 已知中序遍历序列和另外一个序列可以求第三个序列。

前序：$A\,\underbrace{B\,D\,E\,H}_{L}\,\underbrace{C\,F\,G\,I}_{R}$

中序：$\underbrace{D\,B\,H\,E}_{L}\,A\,\underbrace{F\,C\,G\,I}_{R}$

后序：$\underbrace{D\,H\,E\,B}_{L}\,\underbrace{F\,I\,G\,C}_{R}\,A$

- 前序的第一个是 root，后序的最后一个是 root。
- 先确定根节点，然后根据中序遍历，在根左边的为左子树，根右边的为右子树。
- 对于每一个子树可以看成一个全新的树，仍然遵循上面的规律。

BFS遍历

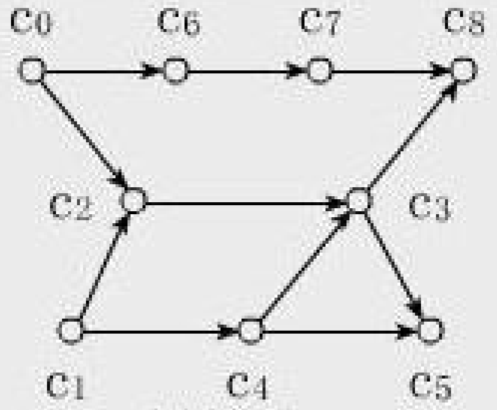
- 与图直接BFS遍历类似，这里不在赘述
- 但是对于树而言，用的基本上都是DFS，BFS使用较少

拓扑排序

Topological sorting

DAG图（有向无环图）

- 边有向，无环。
- 英文名叫 Directed Acyclic Graph，缩写是 DAG。

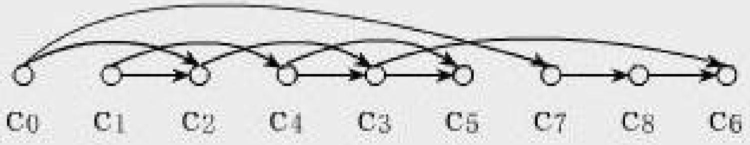


(a) 图G_9

拓扑序列

- 对一个有向无环图G进行拓扑排序，是将G中所有顶点排成一个线性序列，使得图中任意一对顶点u和v，若边(u， v)∈E(G)，则u在线性序列中出现在v之前。

- 通常，这样的线性序列称为满足拓扑次序的序列，简称拓扑序列。



(b)图Ga的拓补次序排列

BFS遍历实现拓扑排序(Kahn 算法)

- 初始状态下，集合S装着所有入度为0的点，L是一个空列表。
- 每次从S中取出一个点u(可以随便取)放入L , 然后将与u相连接的所有边删除。对于边(u, v)，若将该边删除后点v的入度变为0，则将 v 放入S中。
- 不断重复以上过程，直到集合S为空。检查图中是否存在任何边，如果有，那么这个图一定有环路，否则返回L，L中顶点的顺序就是拓扑排序的结果。

BFS遍历实现拓扑排序(Kahn 算法)

```cpp
int main(){
    cin >> n >> m;
    for(int i = 1; i <= m; i++)
{       int u,v; cin >> u >> v;
        G[u].push_back(v);
        in[v] ++;
   }
    toposort();
}
```

```cpp
bool toposort()
{
    vector<int> L;
    queue<int> S;
    for (int i = 1; i <= n; i++)
        if (in[i] == 0)
            S.push(i);
    while (!S.empty())
   {
        int u = S.front();
        S.pop();
        L.push_back(u);
        for (auto v : G[u])
       {
            if (--in[v] == 0)
           {
                S.push(v);
           }
       }
   }
    if (L.size() == n)
    { // 若能对n个排序，则说明不存在环，反之有环，无法确定拓扑序
        for (auto i : L)
            cout << i << ' ';
        return true;
   }
    else
   {
        return false;
   }
}
```

Thanks