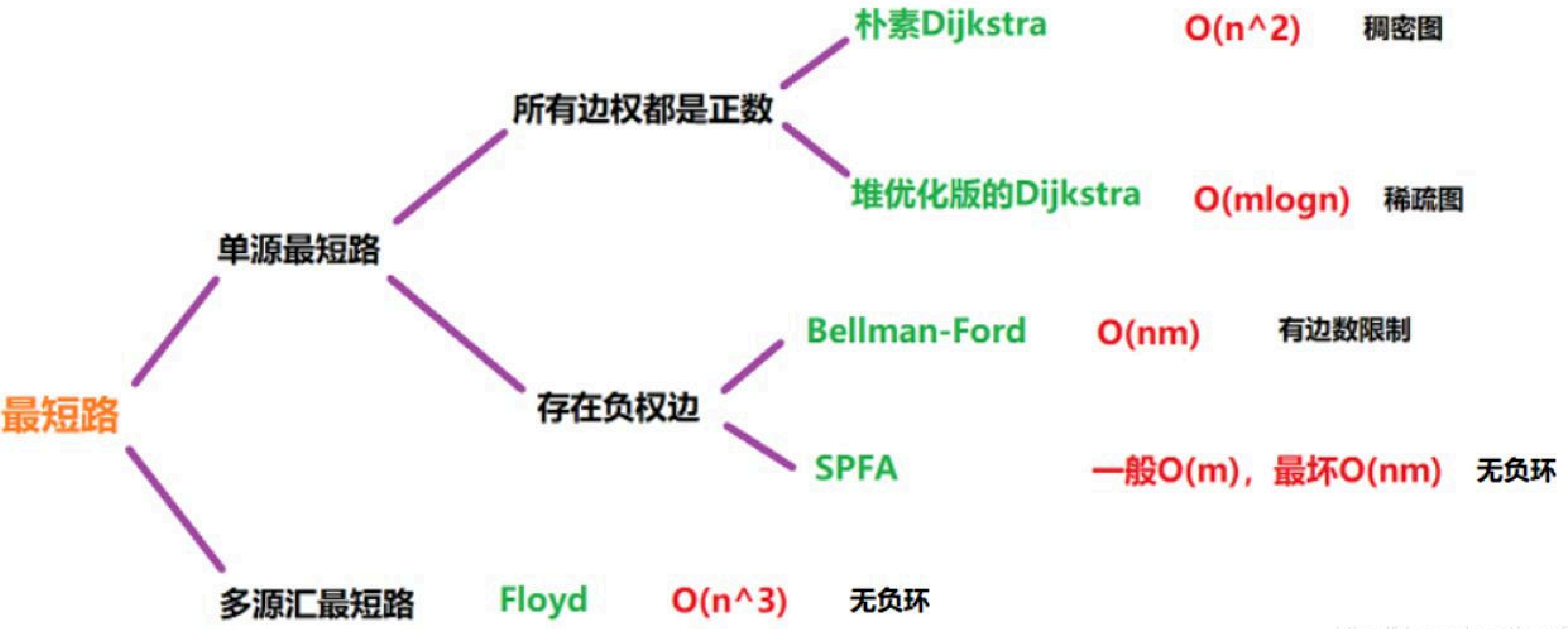
第八章

最短路和拓扑排序算法

授课人：万全　　时间：2024/7/23

Lanzhou University

最短路


最短路
单源最短路
所有边权都是正数
朴素Dijkstra
O(n^2)
稠密图
堆优化版的Dijkstra
O(mlogn)
稀疏图
存在负权边
Bellman-Ford
O(nm)
有边数限制
SPFA
一般O(m)，最坏O(nm)
无负环
多源汇最短路
Floyd
O(n^3)
无负环

单源最短路： 求一个点到其他点的最短路
多源最短路： 求任意两个点的最短路
稠密图用邻接矩阵存，稀疏图用邻接表存储。
稠密图： m 和 n^2 一个级别
稀疏图： m 和 n 一个级别

Dijkstra算法

Dijkstra算法:

1、初始化：dist[1] = 0, dist[i] = +∞　　n为点数，m为边数　　朴素 O(n^2)　　堆优化　O(mlogn)

2、for 循环n次

①在没有确定最短路中的所有点找出距离最短的那个点 t　　总共n^2次 ⟶ 总共n次

②将 t 标记，代表已经找到最短路　　总共n次

③用 t 更新其他点的最短路距离　　总共n^2次 ⟶ 总共mlogn次

集合s：所有已经确定最短路的点

①的意思就是在集合s外找一个距离起点最近的点

②的意思就是让这个最近点放到集合 s 中

Dijkstra算法是通过 n 次循环来确定 n 个点到起点的最短路的。

首先找到一个没有确定最短路且距离起点最近的点，并通过这个点将其他点的最短距离进行更新。每做一次这个步骤，都能确定一个点的最短路，所以需要重复此步骤 n 次，找出 n 个点的最短路。

例 8.2－1 考虑图 8.2－13 中的图，起初 $S=\{a\}$，$T=\{v_1,v_2,v_3,v_4\}$，$D(a)=0$，$D(v_1)=2$，$D(v_2)=+\infty$，$D(v_3)=+\infty$，$D(v_4)=10$。因为 $D(v_1)=2$ 是 T 中最小的 D 值，所以选 $x=v_1$。置 S 为 $S\cup\{x\}=\{a,v_1\}$，置 T 为 $T-\{x\}=\{v_2,v_3,v_4\}$。然后计算：

$$D(v_2)=\min(+\infty,2+3)=5$$

$$D(v_3)=\min(+\infty,+\infty)=+\infty$$

$$D(v_4)=\min(10,2+7)=9$$

如此类推，直至 $T=\varnothing$ 终止。整个过程概括于表 8.2－1 中。

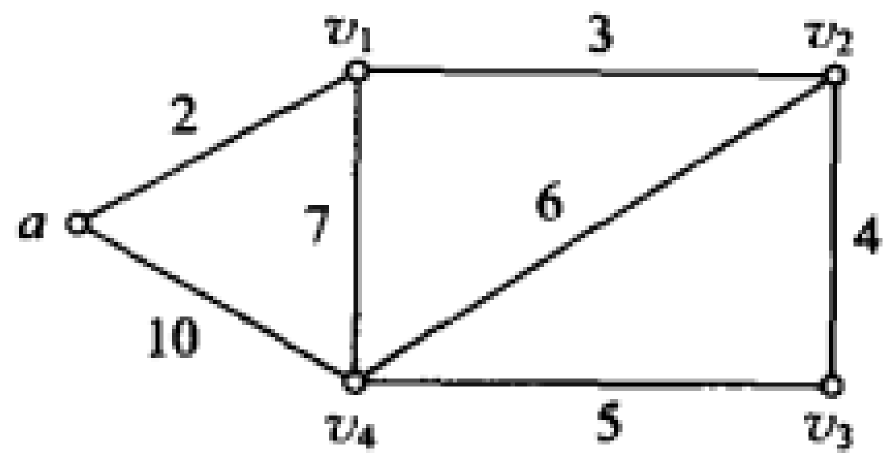


图 8.2－13

+∞=1<<30或0x3f3f3f3f

表 8.2－1

重复次数	S	x	$D(x)$	$D(v_1)$	$D(v_2)$	$D(v_3)$	$D(v_4)$
开　始	$\{a\}$	—	—	2	$+\infty$	$+\infty$	10
1	$\{a,v_1\}$	v_1	2	2	5	$+\infty$	9
2	$\{a,v_1,v_2\}$	v_2	5	2	5	9	9
3	$\{a,v_1,v_2,v_3\}$	v_3	9	2	5	9	9
4	全　部	v_4	9	2	5	9	9

该算法基于“最短路径的任一段子路径都是最短路径”这一事实，所以在算法第(2)条中，在写出最短路径长度 $D(x)$ 的同时，记下最短路径上邻接于 x 的结点名，即可容易求出 a 到所有结点的最短路径。另外，此算法对简单连通有向图也有效。

链式前向星存图

```
int head[N],cnt;
struct edge{
	int v,w,nxt;
}e[M<<1];
void addedge(int u,int v,int w){
	e[++cnt].v=v;e[cnt].w=w;
	e[cnt].nxt=head[u];
	head[u]=cnt;
}
```

```
//记录有哪些节点被松弛
struct node{
    int u,dis;
    friend bool operator <(const node &a,const node &b){
        return a.dis>b.dis;
    }
}tp;
priority_queue<node>q;
```

dijkstra 主函数

```
void dij(){
    memset(dis,0x3f,sizeof dis);
    dis[s]=0;
    q.push((node){s,0});
    while(!q.empty()){
        node now=q.top();q.pop();
        int u=now.u;
        if(now.dis!=dis[u])continue;
        for(int i=head[u];i;i=e[i].nxt){
            int v=e[i].v;
            if(dis[u]+e[i].w<dis[v]){
                dis[v]=dis[u]+e[i].w;
                q.push((node){v,dis[v]});
            }
        }
    }
}
```

例题

洛谷P3371（弱化版）：
https://www.luogu.com.cn/problem/P3371

洛谷P4779（标准版）：
https://www.luogu.com.cn/problem/P4779

Dijkstra算法输出路径

3.5.3.1 路径的记录

如果只需要记录一条最短路径，那么我们在进行松弛操作后用一个数组$pre[n]$记住用于松弛该结点的s_k即可。即：

$$pre[s_j] = s_k$$

我们这里实现了多条最短路径的输出，一维数组不再能满足我们的要求，我们需要一个二维数组$pre[n][m]$记录所有的前驱结点。对于一个结点编号为i的结点，其所有的前驱为$pre[i]$集合中的所有元素。当然，如果最短路径只有一条，那么$pre[i]$中只有一个元素。

另外，需要存储多条最短路径后，在松弛时进行的操作和只输出一条路径时不相同：

- 若$dis[s_k] + weight(s_k, s_j) < dis[s_j]$：

  直接将$pre[s_j]$**置为**s_k的编号，然后将队列中修改s_j的权重为$dis[s_k] + weight(s_k, s_j)$，更新$dis[s_j] = dis[s_k] + weight(s_k, s_j)$

- 若$dis[s_k] + weight(s_k, s_j) == dis[s_j]$

  向$pre[s_j]$中**增加**s_k的编号

路径的输出

按照上述的方式进行记录后，我们最终得到的pre数组，为了便于说明，我们给出下面的一个例子：

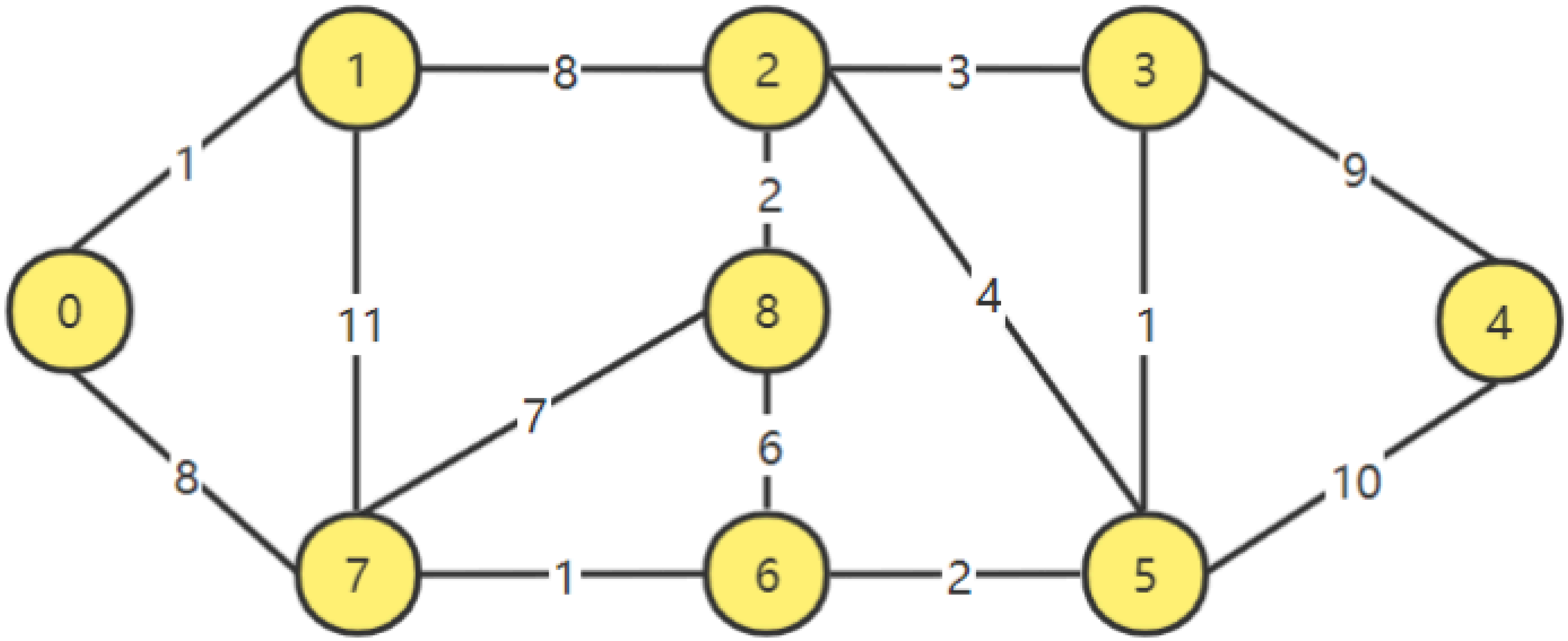


图18：pre数组举例

以图中的结点4为例，我们给出他的pre数组树：

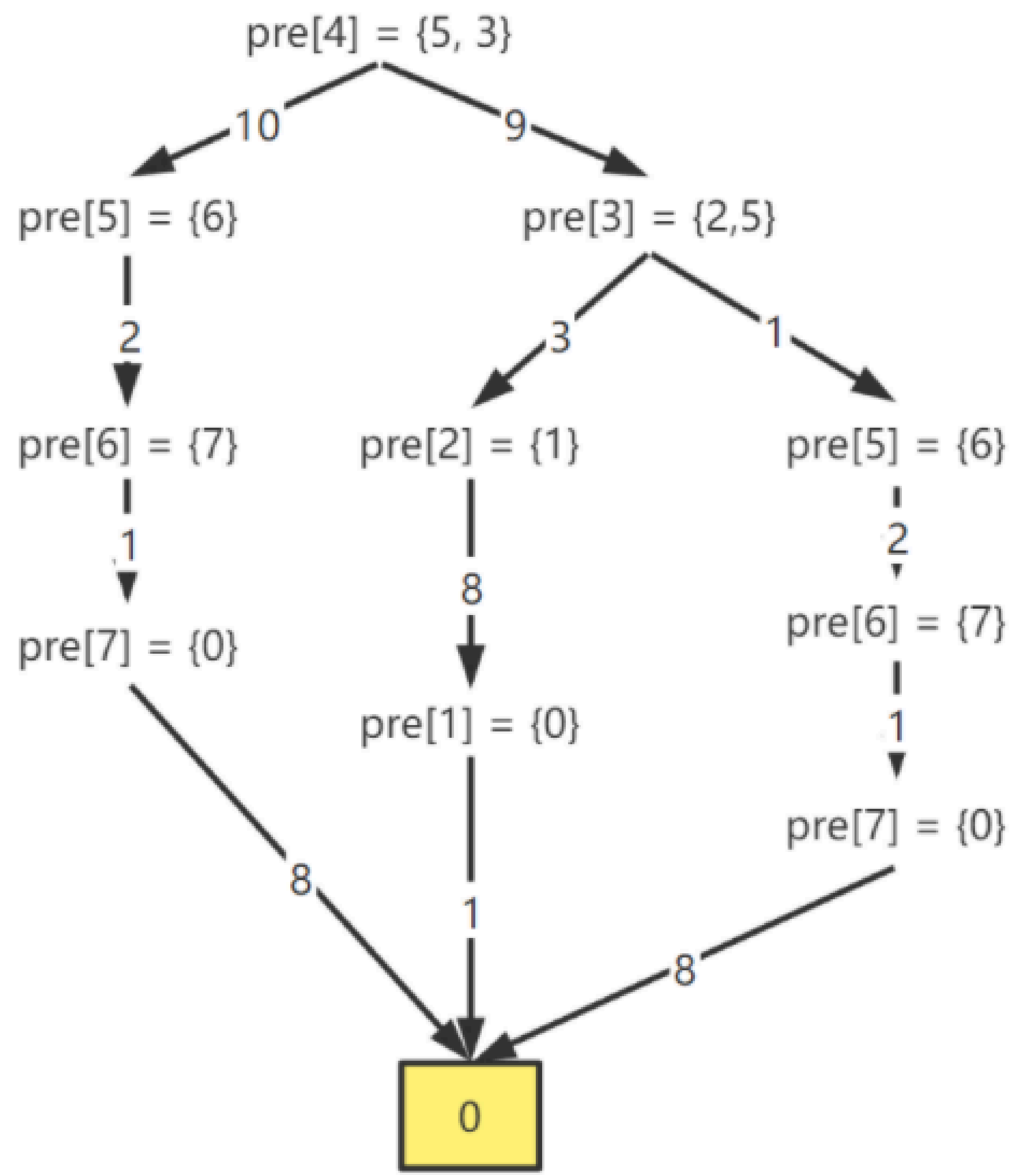


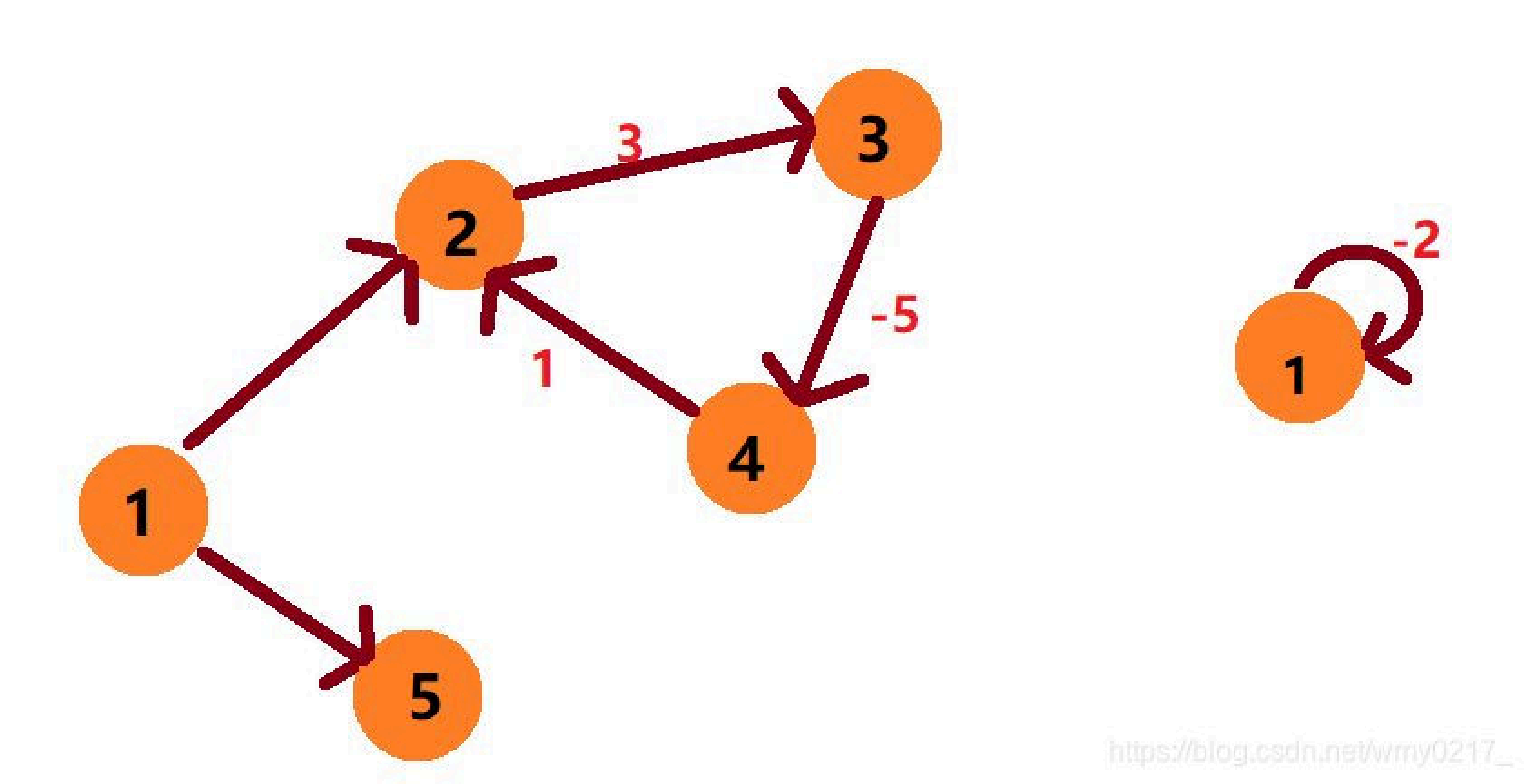
图19：pre数组树

根据这棵pre数组树，我们能够通过深度优先遍历的方式得到从结点4开始到源点0的所有的最短路径。在深搜过程中我们每得到一条路径就应该将他存在一个二维的path数组，每一维存储一条路径。在得到所有最短路径后，遍历path数组的每个元素，每一个元素即是一条路径的路径数组，逐条打印出路径结果即可。

判断负环

什么是负环呢？ 下图左边的2——>3——>4就是一个负环，因为转一圈后的距离是负的，右图的 1 结点是应该自环，也属于负环。

SPFA



相比上一个代码，多了一个cnt数组，cnt[x] 代表起点到x最短路所经的边数，当 cnt[x] ⩾ n 时，则说明 1——>x 这条路径上至少经过 n 条边 ，那么也就是 1——>x 这条路径上至少经过 n+1 个点，而我们知道总共只有 n 个点，说明至少存在两个点是重复经过的，那么这个点构成的环一定是负环，因为只有负环才会让dist距离变小，否则我们为什么要两次经过同一个点呢。

刚开始我们需要让所有点都入队，因为 1 这个结点可能跟我们要找的负环是不连通的，这样的话只通过 1 来是无法判断的，所以，我们让所有结点都入队。

dist数组是否初始化在这里是不影响的，因为我们要求的是是否存在负环，不是距离。

```
bool spfa(){
    queue<int> q;
    for(int i=1; i<=n; i++){ //将所有结点入队
        st[i] = true;
        q.push(i);
    }
    while(q.size()){ // 队列不空
        int t = q.front(); //取队头
        q.pop();
        st[t] = false; // 代表这个点已经不在队列了
        for(int i = h[t]; i!=-1; i=ne[i]){ // 更新 t 的所有临边结点的最短路
            int j = e[i];
            if(dis[j] > dis[t]+w[i]){
                dis[j] = dis[t] + w[i];
                cnt[j] = cnt[t] + 1; // t到起点的边数+1
                if(cnt[j] >= n) return true;// 存在负环
                if(!st[j]){  //如果 j 不在队列，让 j 入队
                    q.push(j);
                    st[j] = true;  // 标记 j 在队中
                }
            }
        }
    }
    return false;// 不存在负环
}
```

Floyd算法

Floyd算法是基于动态规划的，从结点 i 到结点 j 的最短路径只有两种：

1、直接 i 到 j

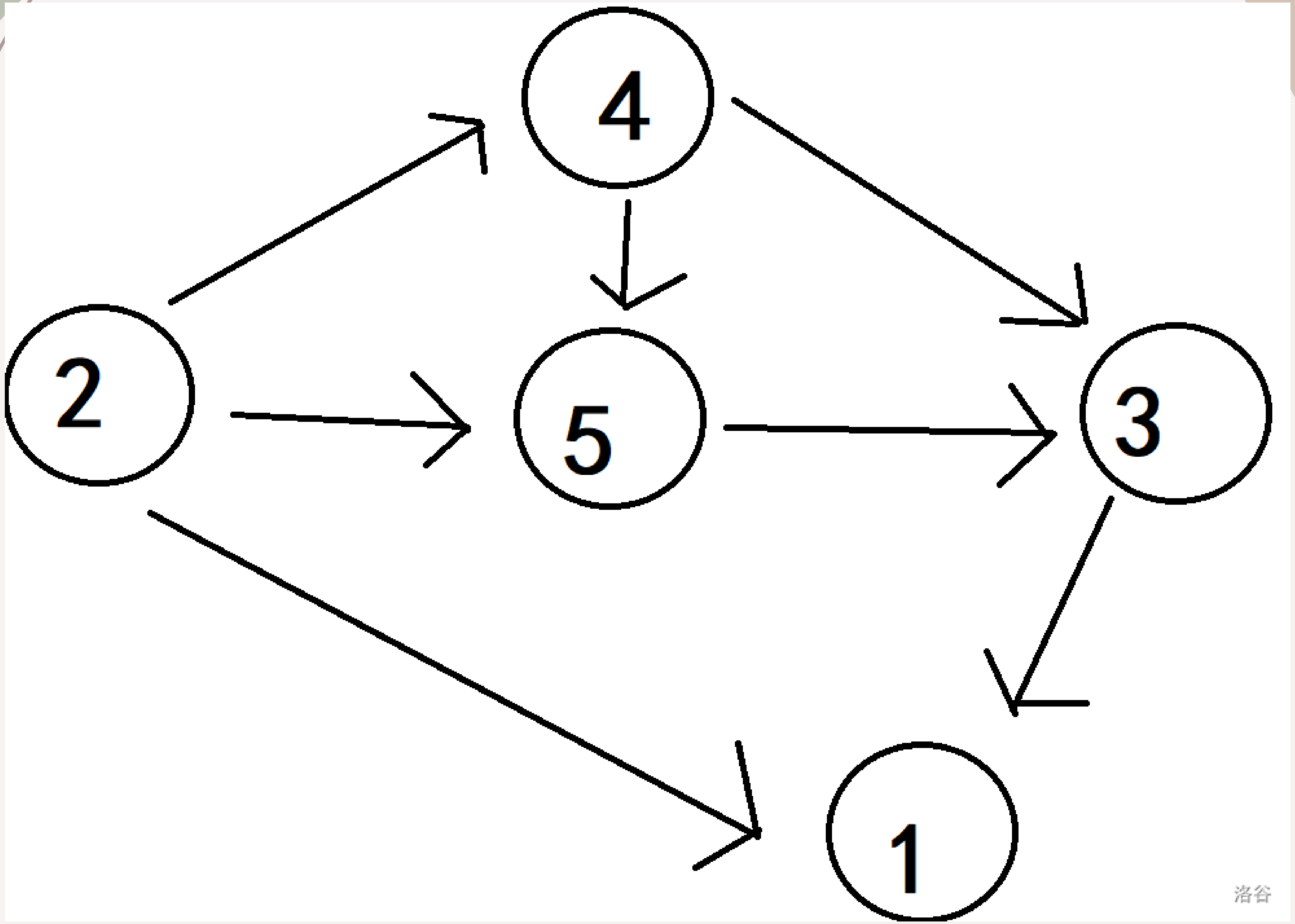
2、i 经过若干个结点到 k 再到 j

对于每一个k，我们都判断 d[i][j] 是否大于 d[i][k] + d[k][j]，如果大于，就可以更新d[i][j]了。

```
void floyd()
{
 for(int k=1; k<=n; k++)
   for(int i=1; i<=n; i++)
     for(int j=1; j<=n; j++)
      d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

拓扑排序

1.构造有向图，记录每个节点的入度（即指向该节点的边的数量）。
2.从图中选择一个入度为0的节点，输出该节点。
3.从图中删除该节点及所有以该节点为起点的有向边。
4.重复上述两步，直到所有节点都被输出，或者当前图中不存在入度为0的节点为止。如果图中不存在入度为0的节点，则说明图中存在环，无法进行拓扑排序。



例题：https://www.luogu.com.cn/problem/B3644

Step1：

```
for(int i=1; i<=n; i++)
        if(d[i]==0)
            q.push(i);  //从图中选择一个入度为0的节点放入队列
```

Step2：

当队列不为空则循环以下操作：取出队首元素并输出，遍历队首元素的连边，对应节点的入度 -1，即删除所有以该节点为起点的有向边，再判断当对应的节点入度为 0 就加入队列。

```
while(q.size())
    {
        x=q.front();    //从队列中取出一个节点
        q.pop();        //将队列中的该节点删除
        cout<<x<<" ";   //输出该节点
        for(int i=1; a[x][i]!=0; i++)
        {
            d[a[x][i]]--;   //从图中删除有以该节点为起点的有向边
            if(d[a[x][i]]==0)
                q.push(a[x][i]);//从图中选择一个入度为0的节点放入队列
        }
    }
```