中国计算机学会
China Computer Federation

图 论 初 步

安徽师大附中　　叶国平

QQ:17412182

讲课的主要内容

- 1.图的基本概念、图的深度优先遍历、图的宽度优先遍历
- 2.Prim和Kruskal算法求最小生成树
- 3.求次小生成树算法
- 4.Floyd-Warshall算法求任意两点间的最短路算法和传递闭包
- 5.Bellman-Ford、Dijkstra、SPFA等求单源最短路算法
- 6.有向无环图的拓扑排序算法
- 7.求最近公共祖先

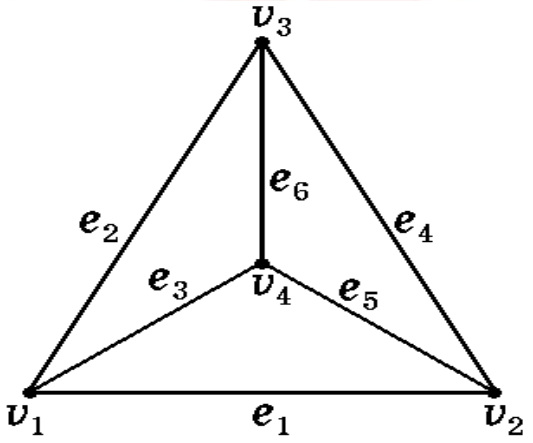
图的基本概念

- 图论中的“图”并不是通常意义下的几何图形或物体的形状图，而是以一种抽象的形式来表达一些确定的事物之间的联系的一个数学系统。
- 一个有序二元组(V, E)称为一个图，记为$G=(V, E)$，其中：
  - V称为G的顶点集，$V\neq\phi$，其元素称为顶点或结点，简称点；
  - E称为G的边集，其元素称为边，它联结V中的两个点，一般用(v_x,v_y)表示，其中$v_x,v_y\in V$。

有向图和无向图

- 如果边连接的两个点是无序的, 则称该边为无向边, 否则, 称为有向边。
- 如果E的每一条边都是无向边, 则称G为无向图(如图1); 如果E的每一条边都是有向边, 则称G为有向图(如图2); 否则, 称G为混合图。

图1



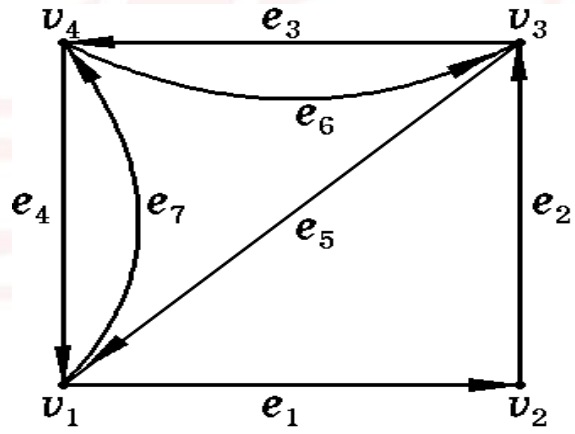


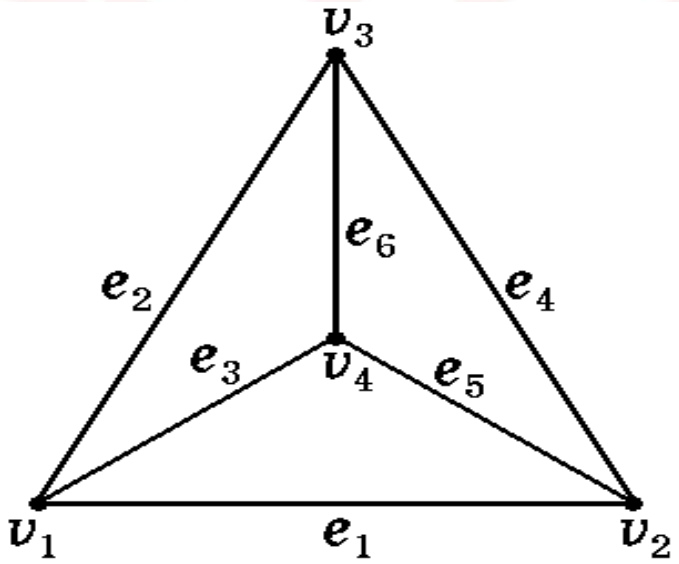
图2

无向图和有向图

- 在无向图中，用一对圆括号表示无向边，显然(v_x,v_y)和(v_y,v_x)是两条等价的边。
- 在有向图中，用一对尖括号表示有向边，把$<v_x,v_y>$中v_x成为起点，v_y称为终点。显然此时的$<v_x,v_y>$和$<v_y,v_x>$是两条不同的边。有向图中的边又称为弧，起点称为弧头，终点称为弧尾。
- 如果两个顶点u、v之间有一条边相连，则称u、v这两个顶点是关联的。

图解

- 对于一个图$G = (V, E)$, 人们常用图形来表示它, 称其为图解。凡是有向边， 在图解上都用箭头标明其方向。例如, 设$V = \{v_1, v_2, v_3, v_4\}$, $E = \{(v_1,v_2), (v_1,v_3), (v_1,v_4), (v_2,v_3), (v_2,v_4),(v_3,v_4)\}$, 则$G = (V, E)$ 是一个有4个顶点和6条边的图, G的图解如下图所示



带权图、度

- 若将图G的每一条边e都对应一个实数$F(e)$, 则称$F(e)$为该边的权, 这个实数可能是距离、费用、时间、电阻等等。这时称图G为赋权图(网络), 记为$G=(V, E, F)$。
- 有边联结的两个点称为相邻的点, 有一个公共端点的边称为相邻边。 边和它的端点称为互相关联. 常用$d(v)$表示图G中与顶点v关联的边的数目, $d(v)$称为顶点v的度数。度为奇数的点称为奇点，度为偶数的点称为偶点。

度

- 在有向图中，把以顶点v为终点的边的数目称为顶点v的入度，把以顶点u为起点的边的数目称为顶点u的出度，出度为0的顶点为终端顶点。
- 定理1：无向图中所有顶点的度之和等于边数的2倍，有向图中所有顶点的入度之和等于所有顶点的出度之和。
- 定理2：任意一个无向图一定有偶数个（或0个）奇点。

完全图和图的稠密性

- 若无向图中任意两个顶点之间都存在着一条边，有向图中任意两个顶点之间都存在着方向相反的两条边，则称此图为完全图。
- n阶的完全有向图含有n*(n-1)条边，n阶完全无向图含有n*(n-1)/2条边，当一个图接近于完全图时，称为稠密图；相反，当一个图的边很少，称为稀疏图。

简单图

- (1) 顶点个数是有限的；
- (2) 任意一条边有且只有两个不同的点与它相互关联；
- (3) 若是无向图，则任意两个顶点最多只有一条边与之相联结；
- (4) 若是有向图，则任意两个顶点最多只有两条边与之相联结；当两个顶点有两条边与之相联结时，这两条边的方向相反。
- 如果某个有限图不满足(2)(3)(4),可在某条边上增设顶点使之满足。

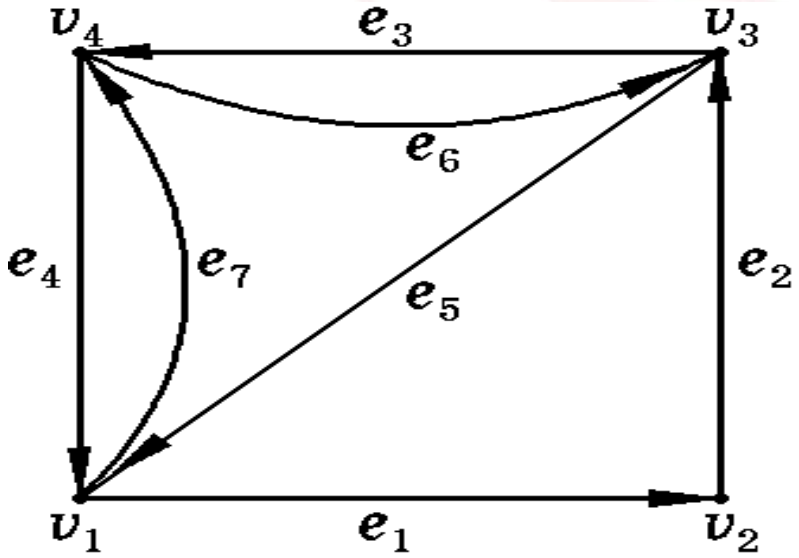
路(径)

- 设$G = (V, E)$是一个图, $v_0, v_1, \ldots, v_k \in V$, 且$\forall i(1 \le i \le k)$，若此图为无向图，则$(v_{i-1}, v_i) \in E$;若此图为有向图，则$<v_{i-1}, v_i> \in E$则，称$v_0\ v_1\ \ldots\ v_k$是$G$的一条通路. 如果通路中没有相同的边, 则称此通路为道路. 始点和终点相同的道路称为环或回路. 如果通路中既没有相同的边, 又没有相同的顶点, 则称此通路为路径, 简称路.
- 路径的长度：路径上的边或弧的数目。

邻接矩阵

- 邻接矩阵：表示了点与点之间的邻接关系。一个n阶图G的邻接矩阵$A=(a_{ij})_{n\times n}$，其中

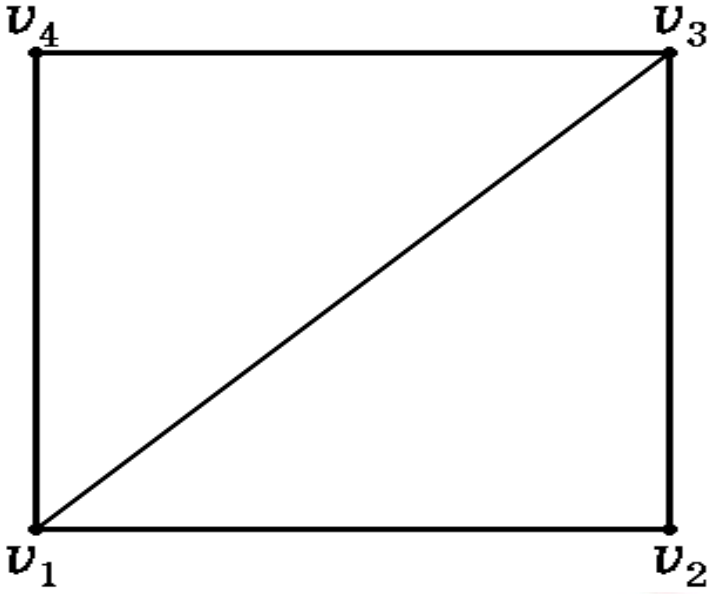
$$a_{ij}=\begin{cases}1, & v_{ij}\in E;\\ 0, & v_{ij}\notin E.\end{cases}$$



$$\boldsymbol{A}=\begin{pmatrix}0&1&0&1\\0&0&1&0\\1&0&0&1\\1&0&1&0\end{pmatrix}$$

邻接矩阵

- 无向图G的邻接矩阵A是一个对称矩阵

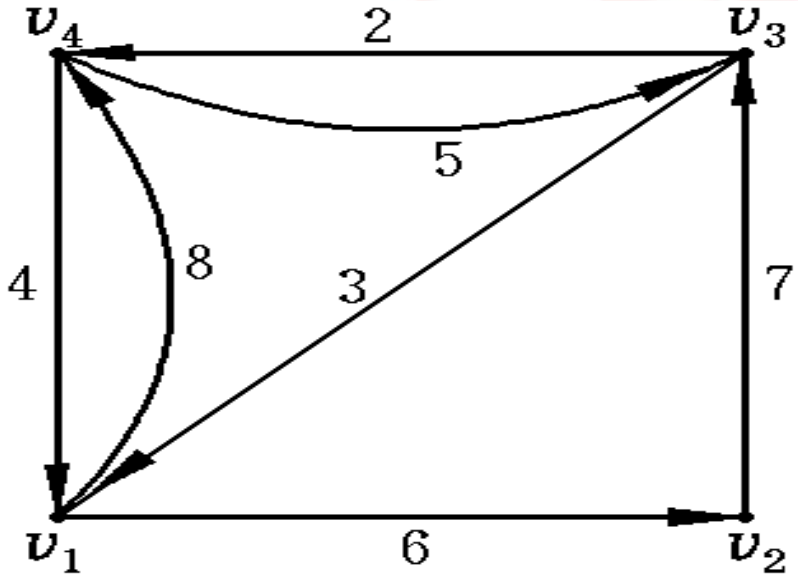


$$\boldsymbol{A}=\begin{pmatrix}0 & 1 & 1 & 1\\ 1 & 0 & 1 & 0\\ 1 & 1 & 0 & 1\\ 1 & 0 & 1 & 0\end{pmatrix}$$

权矩阵

- **权矩阵**：一个n阶赋权图$G=(V,E,F)$的权矩阵$A=(aij)\ n\times n$，其中

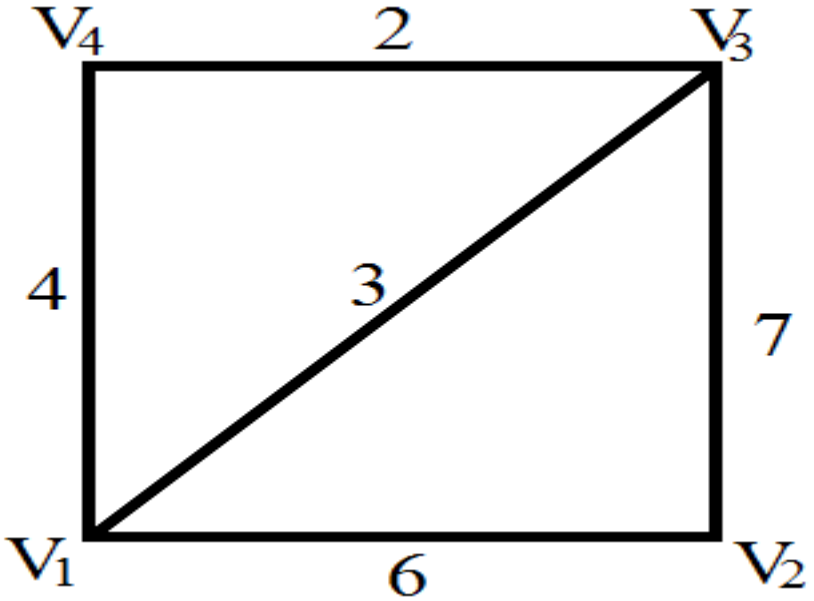
$$a_{ij}=\begin{cases} F(v_iv_j), & v_{ij}\in E; \\ 0, & i=j; \\ \infty, & v_{ij}\notin E. \end{cases}$$



$$\boldsymbol{A}=\begin{pmatrix} 0 & 6 & \infty & 8 \\ \infty & 0 & 7 & \infty \\ 3 & \infty & 0 & 2 \\ 4 & \infty & 5 & 0 \end{pmatrix}$$

权矩阵

- 无向图G的权矩阵A是一个对称矩阵.
- 计算一个顶点的度和邻接点，时间复杂度为O(n).
- 邻接矩阵表示法的空间复杂度为O(n*n),如果用来表示稀疏图，会造成很大的空间浪费。



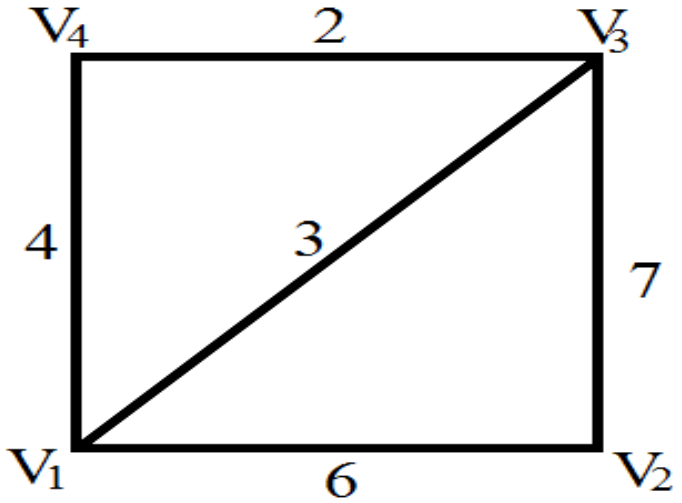
$$A=\begin{pmatrix} 0 & 6 & 3 & 4 \\ 6 & 0 & 7 & \infty \\ 3 & 7 & 0 & 2 \\ 4 & \infty & 2 & 0 \end{pmatrix}$$

边集数值

- 边集数组：利用一维数组存储图中所有边的一种图的表示方法。每个数组元素存储一条边的起点、终点和权值。
- 在边集数组中查找一条边或一个顶点的度都需要扫描整个数组，其时间复杂度为O(e),e为边数。
- 适合对边依次进行处理的运算，而不适合对顶点的运算和对任意一条边的运算，从空间复杂度上讲，边集数组适合于存储稀疏图。

边集数组

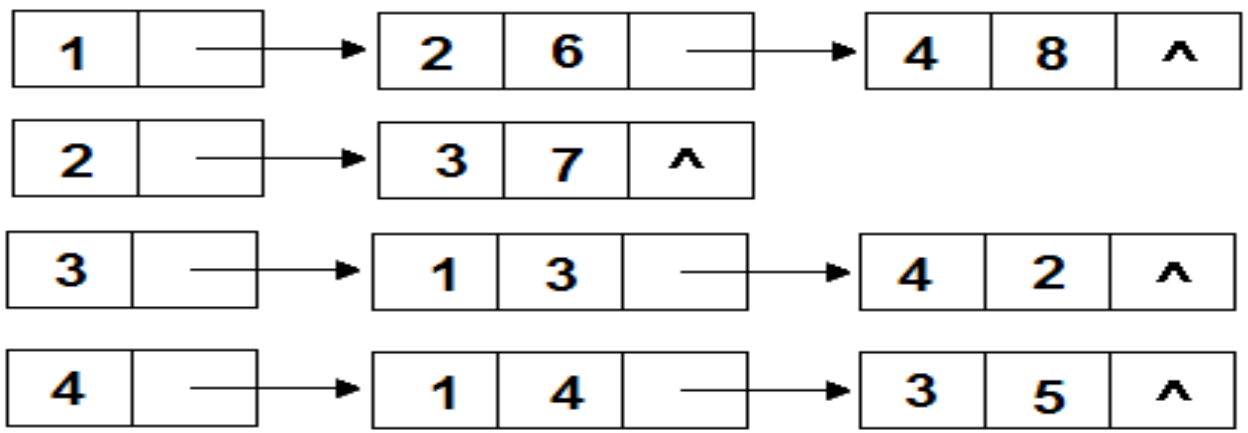
边数	1	2	3	4	5
起点	1	2	3	4	1
终点	2	3	4	1	3
权	6	7	2	4	3

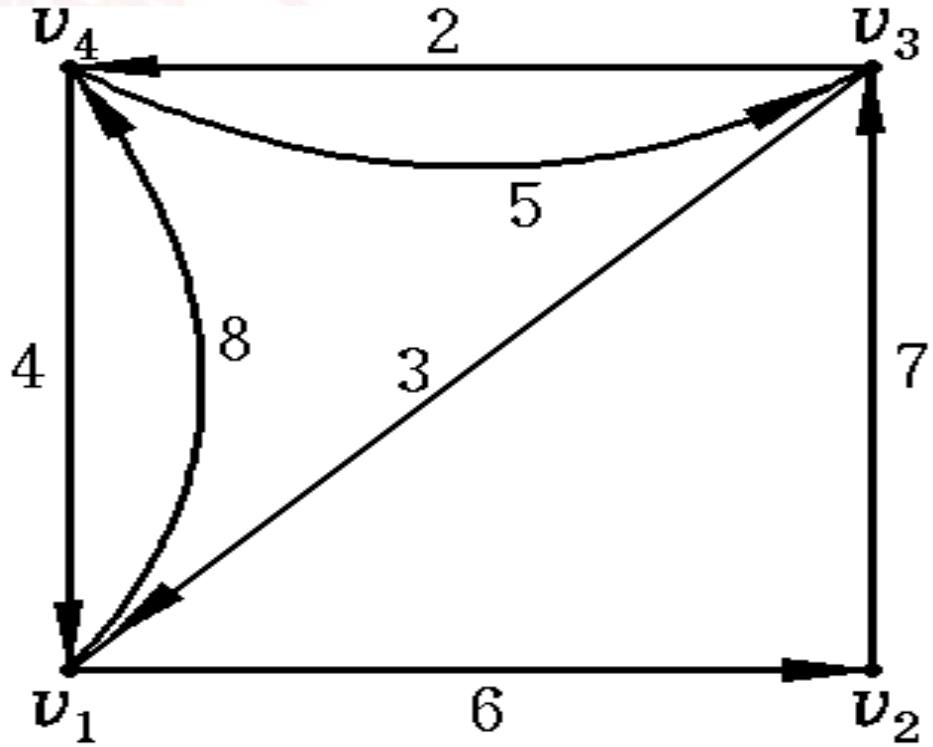


邻接表

- 邻接表：对图中的每个顶点$v_i(1\leq i\leq n)$建立一个邻接关系的单链表,并把它的表头指针用一维向量存储起来的一种图的表示方法.
- 为每个顶点建立的单链表,是表示以该顶点为起点的所有边的信息(包括一个终点序号、一个权值和一个链接域)，一维向量数组除了存储每个顶点的表头指针外，还要存储该顶点的编号i。

邻接表





邻接表

- 自己定义结构体：

```
struct Node //边结点信息
{int adj; //边的终点
 int weight; //权值
 int next; //指向下一条边的链接
}e[10005];
int g[N]; //N个顶点的邻接表:表头
```

- 利用STL中动态数组

```
vector <pair<int,int>> g[N]; //N个顶点的邻接表
```

有向图邻接表的建立

- 自己定义结构体Node的加边程序：

```
void add(int u, int v, int w) //加边,邻接表存储边
{e[++tot].adj=v;  e[tot].weight=w;
  e[tot].next=g[u]; g[u]=tot;
}
```

- 用STL中的vector加边程序：

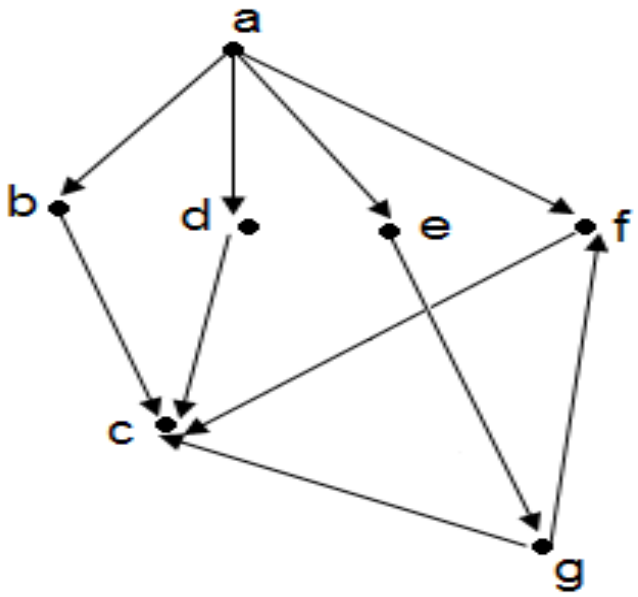
```
for(int u=1; u<=n; u++)
   {int v=read(),w=read();
    g[u].push_back(make_pair(v,w)); //加单向边<u,v>
   }
```

邻接表

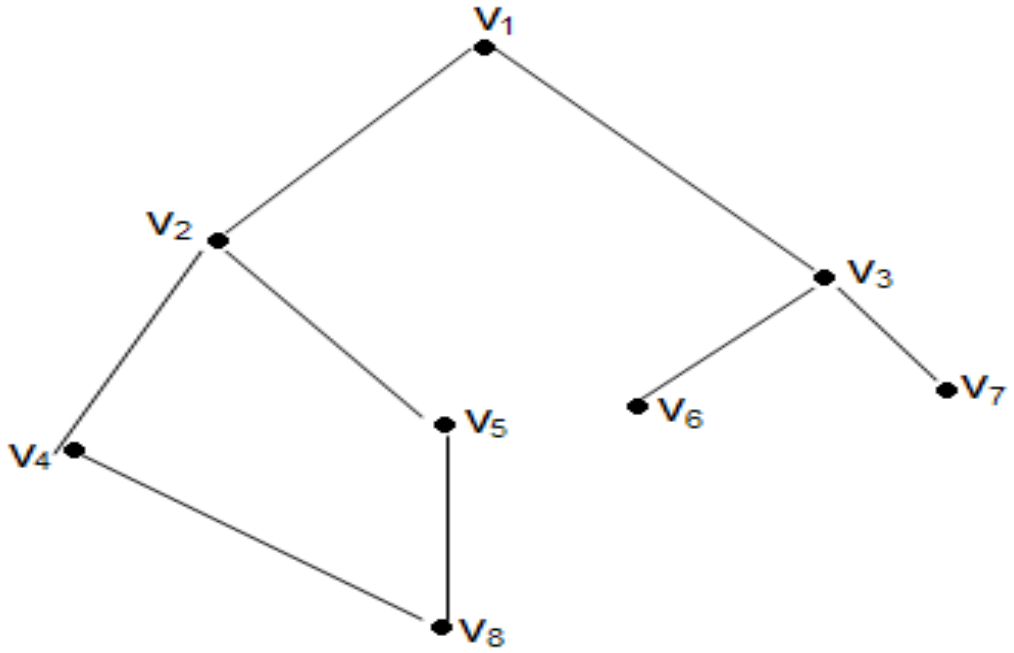
- 邻接表表示法便于查找任一顶点的关联边及邻接点，只要从表头开始进行查找即可。由于无向图的每个顶点的单链表平均长度为2e/n，所以查找时间复杂度为O(e/n)。
- 对有向图来说，想要查找一个顶点的后继顶点和以该顶点为起点的边，包括求该顶点的出度都很容易。但要查找一个顶点的前驱顶点和以此顶点为终点边，以及该顶点的入度就不方便了，需要扫描整个表，时间复杂度为O(n+e)，这时可建一个逆邻接表。

中国计算机学会
China Computer Federation

宽度优先遍历图



a,b,d,e,f,c,g



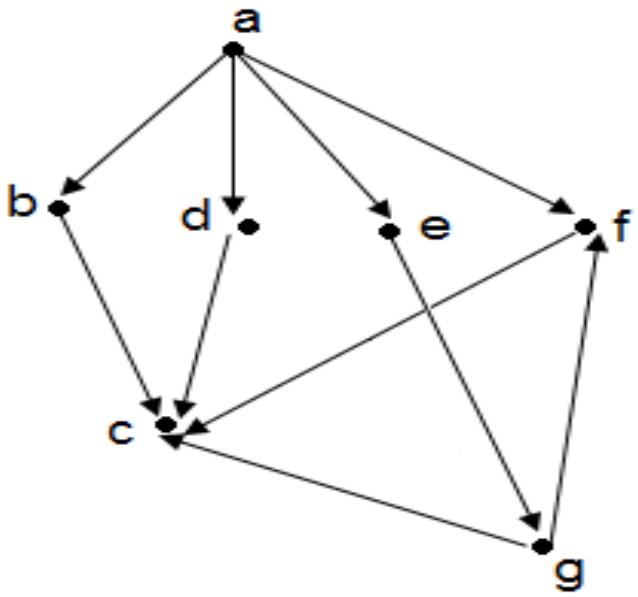
$v_1,v_2,v_3,v_4,v_5,v_6,v_7,v_8$

宽度优先搜索

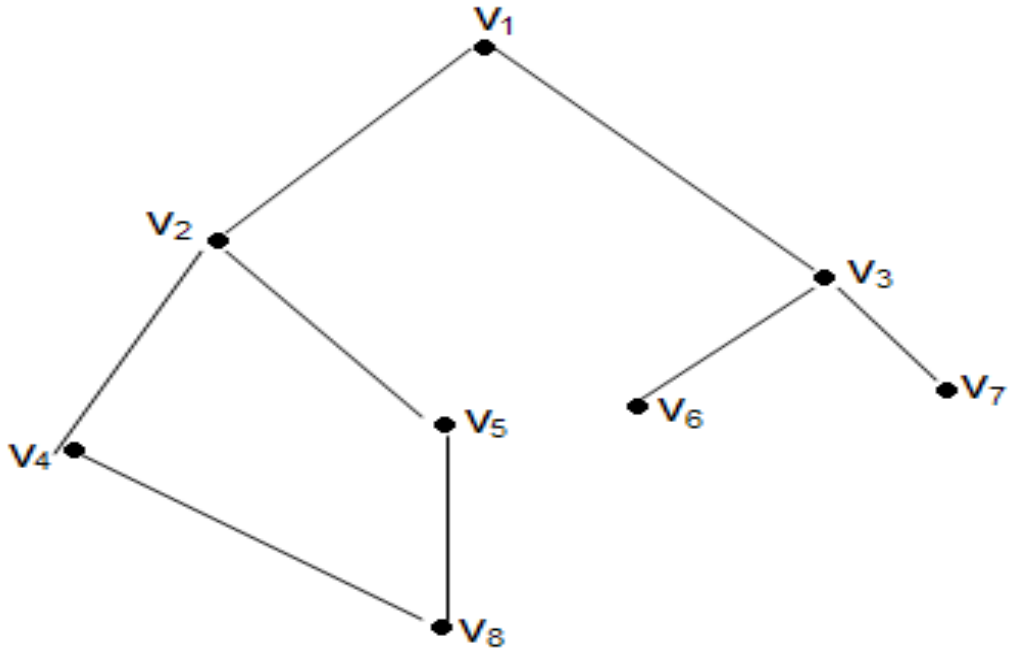
```
void bfs(int s)  //图用邻接表存储
{init(Q); //初始化队列
  printf("%d ",s); visited[s]=true; //访问s并标记
 push(Q, s); //进队列
 while(!empty(Q)) //BFS直到队列为空
   {u=pop(Q); i=g[u]; //取顶点和边表指针
     while (i) //还有后继顶点
       {v=e[i].adj;
        if (!visited[v]) {printf("%d ", v); visited[v]=true; push(Q, v);}//访问v
        i=e[i].next; //从邻接表中取下一个邻接边
       }
   }
}
```

中国计算机学会
China Computer Federation

深度优先遍历图



a,b,c,d,e,g,f



$v_1,v_2,v_4,v_8,v_5,v_3,v_6,v_7$

深度优先遍历图

```
void dfs(int u) //图用邻接表存储
{printf("%d ",u) //输出，最简单的访问方式
 visited[u]=true;
 i=g[u];
 while (i)
   {v=e[i].adj; //v为u的一个后继
    if (!visited[v]) dfs(v);  //深度遍历 递归
    i=e[i].next; //i为i下一个后继
    }
}
```

深度优先遍历(DFS)

- 新发现的结点先扩展
- 得到的可能不是一棵树而是森林, 即深度优先森林(Depth-first forest)
- 特别之处: 引入时间戳(timestamp)
- 发现时间d[v]: 变灰的时间
- 结束时间f[v]: 变黑的时间
- $1 \leq d[v] < f[v] \leq 2|V|$

深度优先遍历(DFS)

- 初始化: time为0, 所有点为白色, dfs森林为空
- 对每个白色点u执行一次DFS-VISIT(u)
- 时间复杂度为O(n+m)

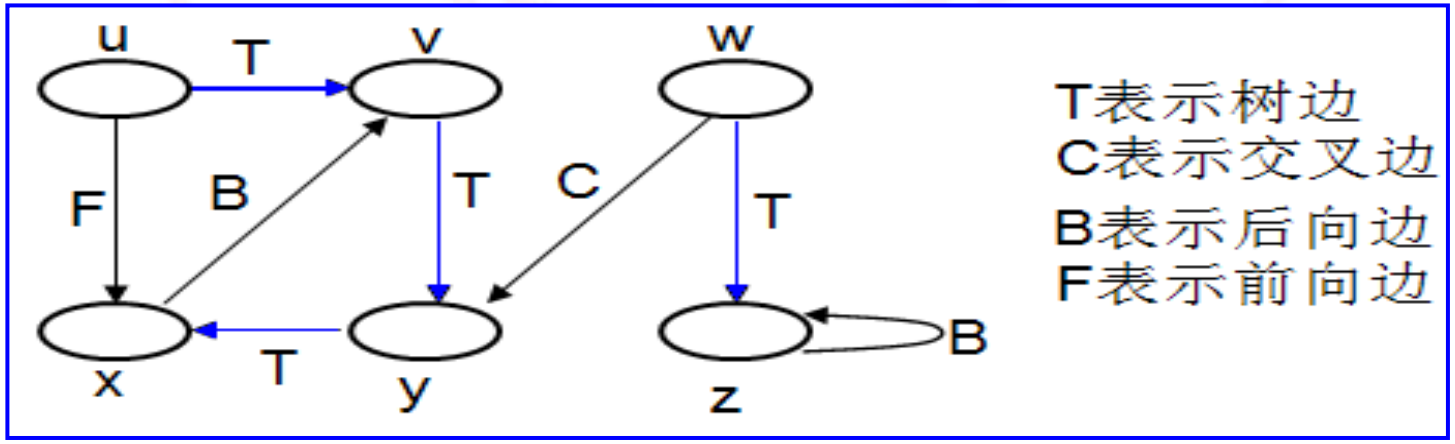
```
DFS-VISIT(u)
1  color[u] ← GRAY            ▷ White vertex u has just been discovered.
2  d[u] ← time ← time + 1
3  for each v ∈ Adj[u]        ▷ Explore edge (u, v).
4       do if color[v] = WHITE
5             then π[v] ← u
6                  DFS-VISIT(v)
7  color[u] ← BLACK           ▷ Blacken u; it is finished.
8  f[u] ← time ← time + 1
```

DFS树的性质

- 括号结构性质
- 对于任意结点对(u, v), 考虑区间[d[u], f[u]]和[d[v], f[v]], 以下三个性质恰有一个成立:
  - 完全分离
  - u的区间完全包含在v的区间内，则在dfs树上u是v的后代
  - v的区间完全包含在u的区间内，则在dfs树上v是u的后代
- 嵌套区间定理：在DFS森林中v是u的后代当且仅当d[u]<d[v]<f[v]<f[u], 即区间包含关系. 由区间性质立即得到。

边的分类

- 一条边(u, v)可以按如下规则分类
  - 树边(Tree Edges, T): v通过边(u, v)发现
  - 后向边(Back Edges, B): u是v的后代
  - 前向边(Forward Edges, F): v是u的后代
  - 交叉边(Cross Edges, C): 其他边，可以连接同一个DFS树中没有后代关系的两个结点，也可以连接不同DFS树中的结点。



边分类算法

- 当(u, v)第一次被遍历, 考虑v的颜色
  - 白色, **(u,v)为T边**
  - 灰色, **(u,v)为B边 (只有它的祖先是灰色)**
  - 黑色: **(u,v)为F边或C边. 此时需要进一步判断**
    - ✓ **d[u]<d[v]: F边 (v是u的后代, 因此为F边)**
    - ✓ **d[u]>d[v]: C边 (v早就被发现了, 为另一DFS树中)**
- 时间复杂度: O(n+m)
- 定理: 无向图只有T边和B边 (易证)

边分类算法

- 实现的细节

```
if  (d[v] == -1) dfs(v);                  //树边,递归遍历
else if (f[v] == -1) show("B");   //后向边
else if (d[v] > d[u]) show("F");  // 前向边
else show("C");                           // 交叉边
```

- d和f数组的初值均为-1, 方便了判断。

例题：Tom猫

- n个节点构成了一棵树，现在需要你给n=a+b个节点进行编号，其中有 a 个节点需要编号为 $1,2,\ldots,a$ 中的不重复的编号表示早餐，剩下的 b 个节点需要编号为 $-1,-2,\ldots,-b$ 中的不重复的编号表示晚餐。
- Tom 每天会随机吃一顿饭，可能是早饭，也可能是晚饭。如果是吃早饭， Tom 会吃掉编号绝对值最小的早餐节点，反之吃掉编号绝对值最小的晚餐节点。
- 如果一个节点被吃掉了，那么与它相连的树上的边都会断掉，因此剩下的节点可能会因此变成若干棵树，即变得不再连通。这是 Tom 不希望发生的事。请给这些香肠编号，使得无论 Tom 如何安排早饭和晚饭，整棵树一直都是连通的

例题：Tom猫

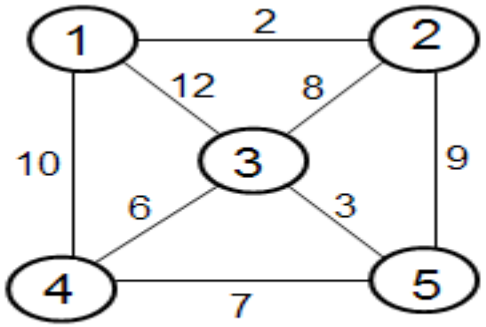
- 第一行三个正整数 n,a,b，代表节点的数目，早餐节点的数目，晚餐节点的数目。保证 $a+b=n$。
  第二行开始，共 $n-1$ 行，每行两个正整数 u,v，代表树上一条边。
- 共 1 行 n 个整数，第 i 个整数为第 i 个节点的编号。
  如果存在多种编号方式，请随意输出一种。如果不存在这样的编号方式，请输出 -1。
- 对于全部的数据， $n\leq 10^5$, $a+b=n$, $1\leq a,b$。

图的最小生成树

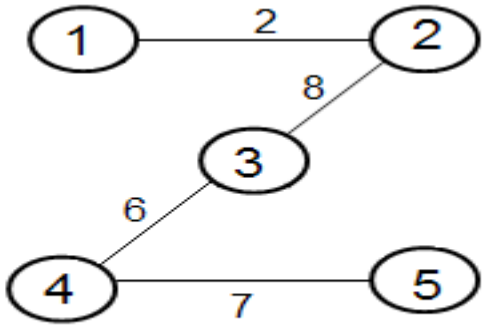
- 如果图G(V,E)是一个连通的无向图，则把它的全部顶点V和一部分边E’构成一个子图G’，即G’=(V,E’),且边集E’能将图中所有顶点连通又不形成回路，则称子图G’是图G的一棵生成树。同一个图可以有不同的生成树，但是：n个顶点的连通图的生成树必定含有n-1条边。
- 对于加权连通图，生成树的权即为生成树中所有边上的权值总和，权值最小的生成树称为图的最小生成树。

图的最小生成树

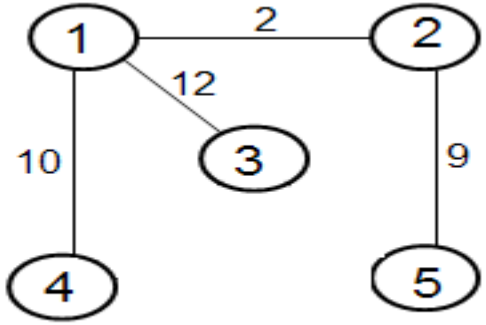
- 求图的最小生成树具有很高的实际应用价值，比如要在若干个城市之间建一个交通网，要求各个城市都能到达且造价最低，这就是一个典型的最小生成树问题。
- 求生成树可以从某个顶点采用深度优先遍历的方法，也可以采用广度优先遍历的方法，分别称为深度优先生成树和广度优先生成树。



(A)



(B)

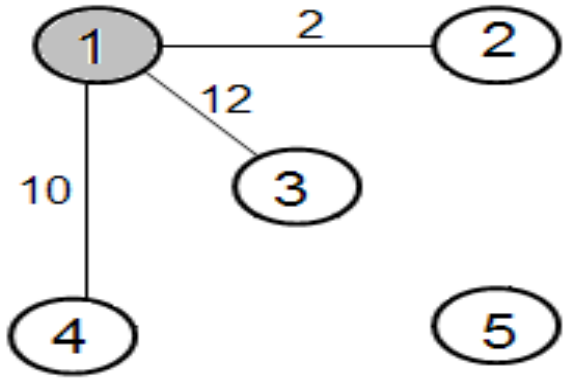


(C)

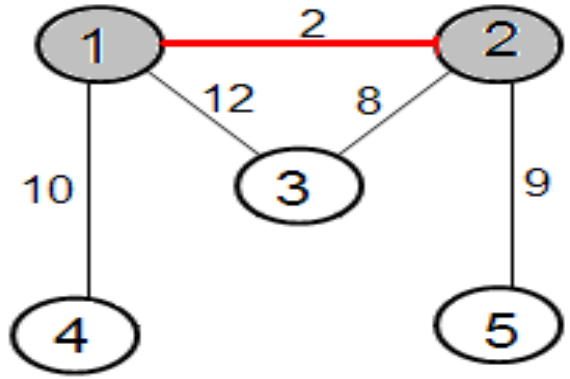
Prim算法

- 设图的阶为n，G=(V,E):
  1. 设置一个顶点的集合S和一个边的集合TE，S和TE的初始状态均为空集；
  2. 选定图中的一个顶点K，从K开始生成最小生成树，将K加入到集合S中；
  3. 重复下列操作，直到选取了n-1条边：选取一条权值最小的边(X,Y)，其中X∈S且!(Y∈S)；将顶点Y加入集合S,边(X,Y)加入集合TE;
  4. 得到最小生成树T=(S,TE)。

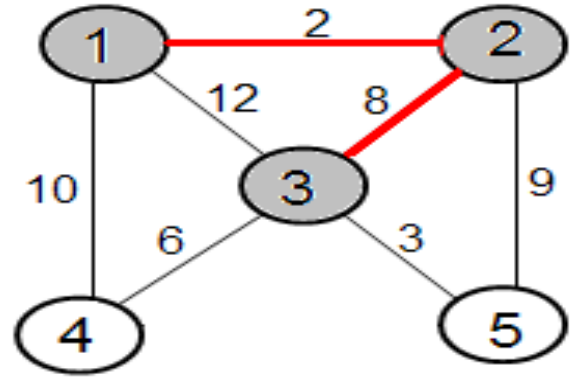
Prim算法



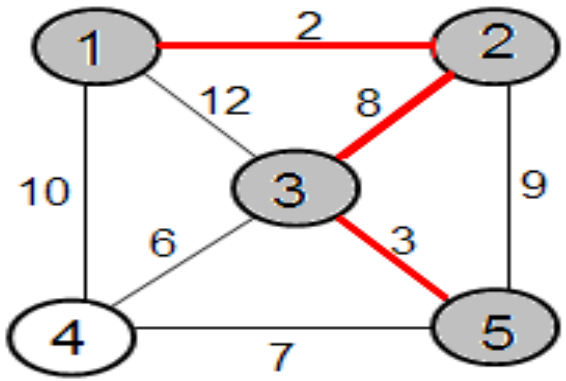
从1开始，S={1}
TE={}



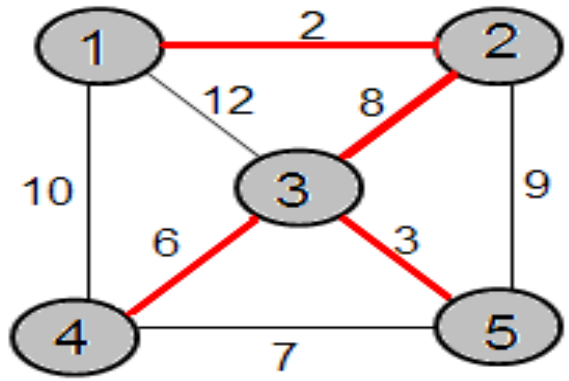
S={1,2}
TE={(1,2)}



S={1,2,3}
TE={(1,2) (2,3)}



S={1,2,3,5}
TE={(1,2) (2,3) (3,5)}



S={1,2,3,4,5}
TE={(1,2) (2,3) (3,5) (3,4)}

Prim算法

- 任意时刻的中间结果都是一棵树
  - 从一个点开始
  - 每次都花最小的代价，用一条边加进一个新点
- 问题：
  - 这样做是对的吗？
  - 如何快速找到这个“最小代价”？

快速找到最小代价

- 需要借助数据结构！
- 我们的算法要求
  - **快速取/删除最小值（边权）**
  - **允许插入边（加入新点时插入它的关接边）**
  - **抽象数据类型：优先队列！**
- 经典实现：堆！

Prim算法框架

初始化，树仅含一个任意一点v_0

把v_0的邻边插入堆

```
for (i=1;i<=n-1;i++)
  {从堆中取出最小值，设边为(u',v')，v'为新点
   (u',v')加入生成树中
   v'和它所有不在树中的邻居组成的边插入堆
  }
```

- 每次取最小值为O(logV)
- 总时间复杂度为O((V+E) log(V)) = O(Elog(V))

Kruskal算法

- 设图的阶为n，G=(V,E)
- 设最小生成树为T=(V,TE)，步骤为：

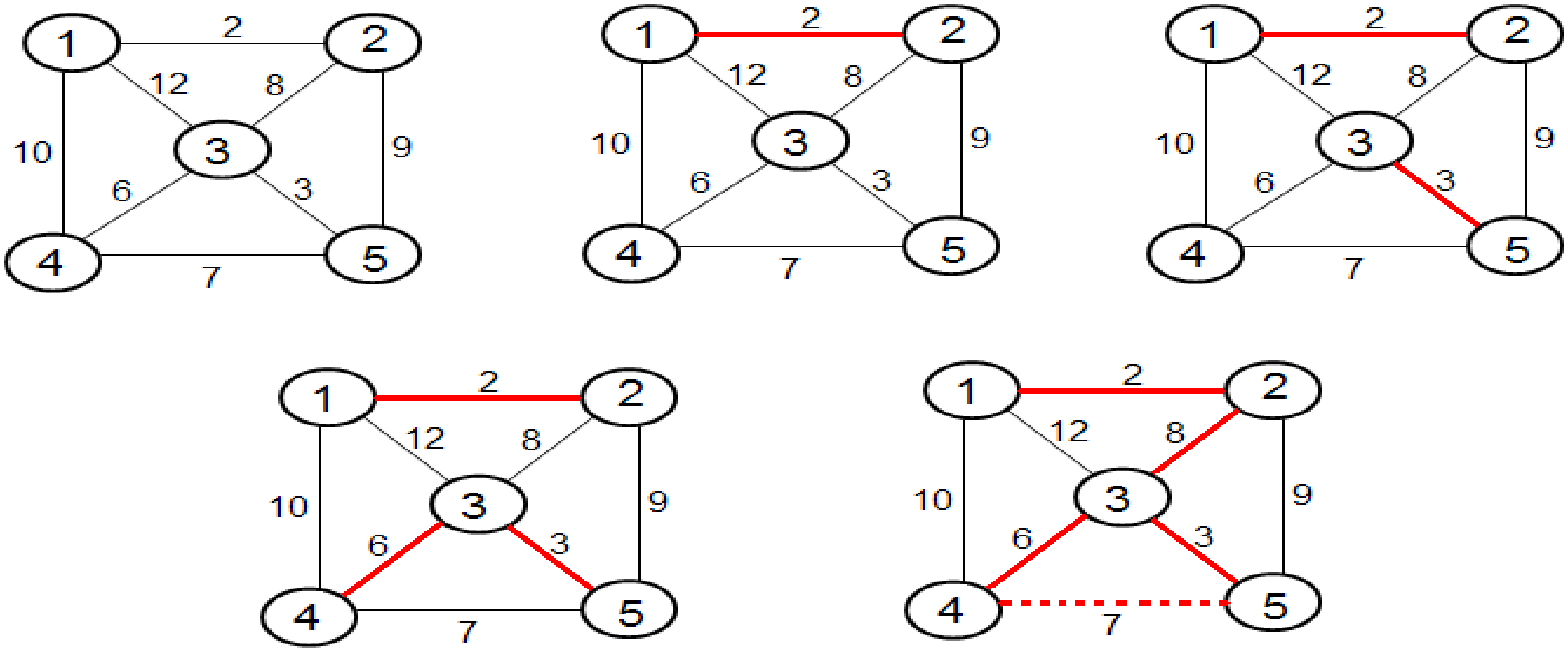
① 设置边的集合TE的初始状态为空集；

② 将G中的边按权值从小到大排好序；

③ 重复下列操作,直到TE中包含n-1条边：从权值小的边开始依次选取，若选取的边使生成树T不形成回路，则把它加入TE中，保留作为T的一条边；若选取的边使生成树形成回路，则将其舍弃；

④ 最后的T即为最小生成树。

Kruskal算法


1 2 2 12 8 10 3 9 6 3 4 7 5
1 2 2 12 8 10 3 9 6 3 4 7 5
1 2 2 12 8 10 3 9 6 3 4 7 5
1 2 2 12 8 10 3 9 6 3 4 7 5
1 2 2 12 8 10 3 9 6 3 4 7 5

Kruskal算法

- 任意时刻的中间结果是一个森林
  - 从n个点的集合开始
  - 每次选不产生圈的前提下权最小的边加入
- 问题：
  - 这样做是对的吗？
  - 如何快速的判断是否产生圈

快速判断是否产生圈

- 我们的算法要求
  - 判断两个点是否在同一棵树中
    - ✓产生圈当且仅当此边连接同一树中的点！
  - 快速把两棵树合并
    - ✓加边意味着两棵树合为一棵
  - 抽象数据类型：并查集！
    - ✓经典实现：森林
- 并查集的森林实现
  - 森林中的每棵树表示不同的集合
  - 树的形态并不重要，有意义的只是“哪些元素在树中”

Kruskal算法框架

把所有边按权值从小到大排序为$e_1,e_2,\ldots$
初始化n个集合，$S_i=\{i\}$
size= 0;
for (i=1;i<=m;i++)
  if e_i的两个端点u, v不在同一个集合 then
    {合并S_u和S_v
     size++;
     if (size == n – 1）break;
    }

- 最坏情况循环执行m次，判断约O(1)，如果输入已经排序好，则总时间复杂度为O(m)，否则为O(mlogm)

次小生成树

- 第一种"**换边**"算法
  - 用**Kruskal**求最小生成树，标记用过的边。
  - 求次小生成树时，依次枚举用过的边，将其去除后再求最小生成树，得出所有情况下的最小的生成树就是次小的生成树。
  - 可以证明：最小生成树与次小生成树之间仅有一条边不同。
- 这样相当于运行(n-1)次Kruskal算法。
- 复杂度O(mlogm+n*m)

次小生成树

- 第二种"最长边"算法
  - 先加入(x,y)，对于一棵树，加入(x,y)后一定会形成环;
  - 如果删去环上除(x,y)外最大的一条边，会得到加入(x,y)时权值最小的一棵树;
  - 如果能快速计算最小生成树中点x与点y之间路径中最长边的长度，即可快速解决。
- 复杂度O(mlogm+n^2)

例题：最小生成树

- 若一个图的每一对不同顶点都恰有一条边相连，则称为完全图。
- 最小生成树MST在Smart的指引下找到了你，希望你能帮它变成一个最小完全图（边权之和最小的完全图）。
  注意：必须保证这个最小生成树MST对于最后求出的最小完全图是唯一的。
- 100%的数据：$n \leq 20000$，所有的边权$\leq 2^{31}$。

例题：灌溉

- Smart决定把水源引到他的$n(1\le n\le 300)$块农田进行灌溉，农田的编号1到n。他将水源引入某一块农田进行灌溉有两种方法：一是在这块农田打一口井，另一个是将这块农田与另一个已经有水源的农田用一根管道连接引水。
- 打一口井需要花费$w[i](1\le w[i]\le 100000)$，把农田$i$与农田$j$用一根管道相连的费用$p[i][j](1\le p[i][j]\le 10^5$，$p[i][j]=p[j][i]$，$p[i][i]=0)$。
- 你计算Smart最少要花多少钱才能够让他的所有农田都有水源进行灌溉。

例题：灌溉

- 第1行：一个数n；
  第2行到第n+1行：第i+1行含有一个数$w[i]$；
  第n+2行到第2n+1行：第n+1+i行有n个被空格分开的数，第j个数代表$p[i][j]$。
- 只有一行，一个单独的数代表最小花费。

积水

输入样例：

```
4
5
4
4
3
0 2 2 2
2 0 3 3
2 3 0 4
2 3 4 0
```

输出样例：

```
9
```

例题：积水

- 有一块矩形土地被划分成 $n \times m$ 个正方形小块。这些小块高低不平，每一小块都有自己的高度。水流可以由任意一块地流向周围四个方向的四块地中，但是不能直接流入对角相连的小块中。一场大雨后，由于地势高低不同，许多地方都积存了不少降水。给定每个小块的高度，求每个小块的积水高度。
- 注意：假设矩形地外围无限大且高度为 0。
- 100%的数据：$n,m \leq 300$，|小块高度|$\leq 10^9$，在每一部分数据中，均有一半数据保证小块高度非负。

最短路问题

- 问题描述：给带权图G=(V,E)
  - **SSSP：求给定起点s(源点)到其他所有点的最短路**
  - **APSP：求每两对点的最短路**
- 算法
  - 动态规划类：**floyd-warshall**
  - 标号设定类：**dijkstra**
  - 标号修正类：**bellman-ford**
  - 变形与应用举例

Floyd-Warshall

- Floyd–Warshall（简称Floyd算法）是一种著名的解决任意两点间的最短路径(APSP)的算法。从表面上粗看，Floyd算法是一个非常简单的三重循环，而且纯粹的Floyd算法的循环体内的语句也十分简洁。我认为，正是由于“Floyd算法是一种动态规划算法”的本质，才导致了Floyd算法如此精妙。因此，这里我将从Floyd算法的状态定义、动态转移方程以及滚动数组等重要方面，来简单剖析一下图论中这一重要的基于动态规划的算法——Floyd算法。

Floyd-Warshall

- 设d[k][i][j]定义成：只能使用第1号~第k号点作为中间媒介时，点i到点j之间的最短路径长度。
- 则它有两种情况进行转移：
  - 如果最短路经过点***k***，那么

    d[k][i][j]=d[k-1][i][k]+d[k-1][k][j]
  - 如果最短路不经过点***k***，那么

    d[k][i][j]=d[k-1][i][j]。
  - 综合起来

    d[k][i][j] = min(d[k-1][i][j], d[k-1][i][k]+d[k-1][k][j])

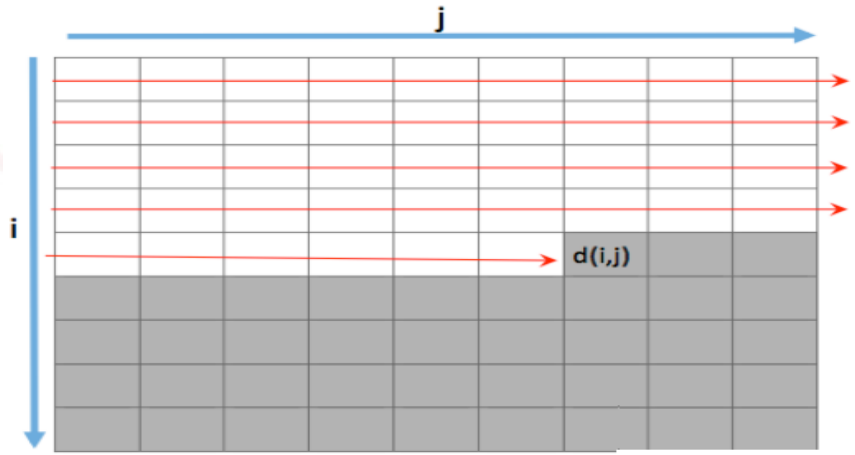
Floyd-Warshall

- 初始化： d[0][i][j]=w[i][j]; //就是边的权值
- 目标答案：d[n][i][j]就是所要求的图中所有的两点之间的最短路径的长度。
- 再次观察动态转移方程:

  d[k][i][j] = min(d[k-1][i][j], d[k-1][i][k]+d[k-1][k][j])
- 可以发现每一个第k阶段的状态(d[k][i][j])，所依赖的都是前一阶段（即第k-1阶段）的状态（如d[k-1][i][j]，d[k-1][i][k]和d[k-1][k][j]）。
- 用滚动数组：d[i][j] = min(d[i][j], d[i][k]+d[k][j])

Floyd-Warshall

- 使用滚动数组，在第k阶段，下图中，白色的格子，代表最新被计算过的元素（即第k阶段的新值），而灰色的格子中的元素值，其实保存的还是上一阶段（即第k-1阶段）的旧值。
- 所以：右侧括号里的d[i][j]保留的值的确就是上一阶段的旧值



使用滚动数组 d[][]（第 k 阶段）

Floyd-Warshall

- 但d[i][k]和d[k][j]呢？我们无法确定这两个元素是落在白色区域（新值）还是灰色区域（旧值）。好在有这样一条重要的性质，dp[k-1][i][k]和dp[k-1][k][j]是不会在第k阶段改变大小的。也就是说，凡是和k节点相连的边，在第k阶段的值都不会变。如何简单证明呢？我们可以把j=k代入之前的d[k][i][j]=min(d[k-1][i][j], d[k-1][i][k] + d[k-1][k][j])方程中，即：d[k][i][k]= min(d[k-1][i][k], d[k-1][i][k]+d[k-1][k][k])== min(d[k-1][i][k], d[k-1][i][k]+0)= d[k-1][i][k]

Floyd-Warshall

- 也就是说在第k-1阶段和第k阶段，点i和点k之间的最短路径长度是不变的。相同可以证明，在这两个阶段中，点k和点j之间的的最短路径长度也是不变的。因此，对于使用滚动数组的转移方程d[i][j] = min(d[i][j], d[i][k]+d[k][j])来说，赋值号右侧的d[i][j], d[i][k]和d[k][j]的值都是上一阶段（k-1阶段）的值，可以放心地被用来计算第k阶段时d[i][j]的值。

Floyd-Warshall

- 基本的动态规划

```
for (k=1;k<=n;k++)
  for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
      {if (i==k || i==j || k==j) contiue;
       if (d[i,k]+d[k,j] < d[i,j]) d[i,j] = d[i,k]+d[k,j];
      }
```

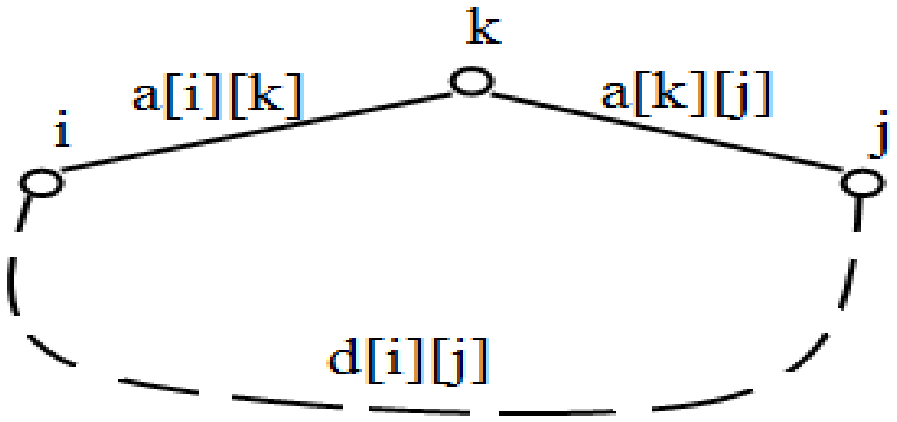
- 很简单，也很好记！
- 时间复杂度：$O(n^3)$

例题：观光旅游

- 某旅游区里面有N个（N≤100）景点。两个景点之间可能直接有道路相连，用a[i][j]表示它的长度，否则它们之间没有直接的道路相连。
- 旅游区规定：每个游客的旅游线路只能是一个回路（好霸道的规定）。也就是说，游客可以任取一个景点出发，依次经过若干个景点，最终回到起点。一天，Smart决定到这个景区来旅游，由于他实在已经很累了，于是他决定尽量少走一些路。
- 他想请你帮他求出最优的路线。

分析

- 这道题要充分理解Floyed的算法思想。
- 如下图：可以设计回路为i → k→ j →...(没有经过点k) → i
- 用a[i][j]表示i和j直接连边的长度，d[i][j]表示i到j的最短路径，ans=min(ans, a[i][k]+a[k][j]+d[i][j])，此时计算的d[i][j]不能经过k点。



程序

```
for (k=1; k<=n; k++)  //枚举中间点
  {for (i=1; i<=n; i++)
     for (j=1; j<=n; j++)
       {if (i==j || i==k || j==k) continue;
        if (ans > dist[i][j] + a[i][k] + a[k][j]) //回路为i->k->j->...->i
          ans = dist[i][j] + a[i][k] + a[j][k];
       }
   for (i=1; i<=n; i++) //floyd求最短路
     for (j=1; j<=n;  j++)
       {if (i==j || i==k || j==k) continue;
        if (dist[i][j] > dist[i][k] + dist[k][j])
          dist[i][j] = dist[i][k] + dist[k][j];
       }
  }
```

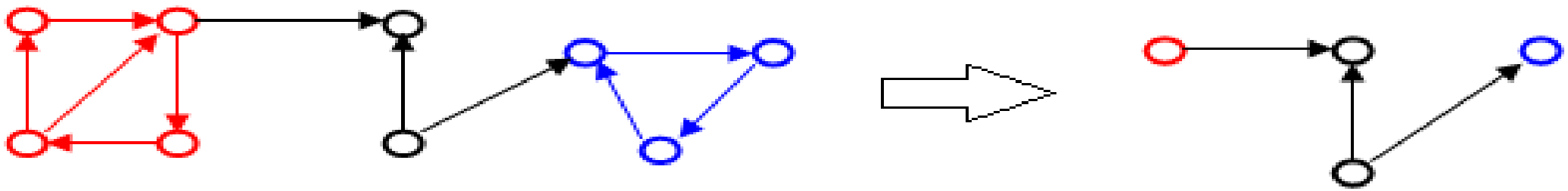
例题：舞会邀请

- Smart准备邀请n（n≤100）个已经确定的人来参加舞会，可是问题来了：n个人每一个人都有一个小花名册，名册里面写着他能够通知到的人的名字。比如说在A的人名单里写了B，那么表示A能够通知到B；但是B的名单里不见得有A，也就是说B不见得能够通知到A。
- Smart觉得需要确定自己需要通知到多少个人（人数m），能够实际将所有n个人都通知到。并求出一种方案以确定m的最小值是多少。

  注意：自己的名单里面不会有自己的名字。

分析

- 本题建模后图是有向图。
- 将图中的强连通分量找出来缩点后重新构图，因为N≤100，可以用Floyed算法求强连通分量。
- Smart需要通知的人数即为重新构图后图中入度为0的顶点个数。



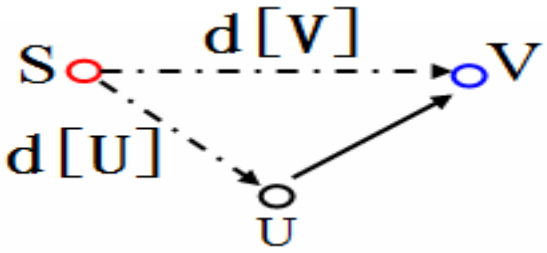
程序

```
for (k=1; k<=n; k++) //用floyed求强连通分量
   for (i=1; i<=n; i++)
     for (j=1; j<=n; j++)
        {if (i==j || j==k || i==k) continue;
          if (a[i][k] && a[k][j]) a[i][j]=true;
        }
 for (i=1, tot=0; i<=n; i++) //缩点后图中点的个数
   if ( !flag[i] )  //flag[i]表示i是否已经分配给一个强连通分量
     {p[i]=++tot; flag[i]=true;
      for (j=i+1; j<=n; j++)
        if (a[i][j] && a[j][i])  //j与i属于同一个强连通分量
          {flag[j]=true;  p[j]=tot; }
     }
```

松弛操作

- 在单源最短路中，设d[v]表示起点S到结点v的当前最短路，pre[v]表示S到v的当前最短路径中v点之前的一个点的编号。

```
Relax(u, v, w)
if(d[v]>d[u]+w(u,v))
   {d[v] = d[u]+w(u,v);
    pre[v] = u;
   }
```



- 松弛操作是改变最短路径和前趋的唯一方式。

Dijkstra算法

- Dijkstra算法是求单源最短路算法，采用的是一种贪心的策略。数组d[ ]来保存源点s到各个顶点的最短距离，初始时:

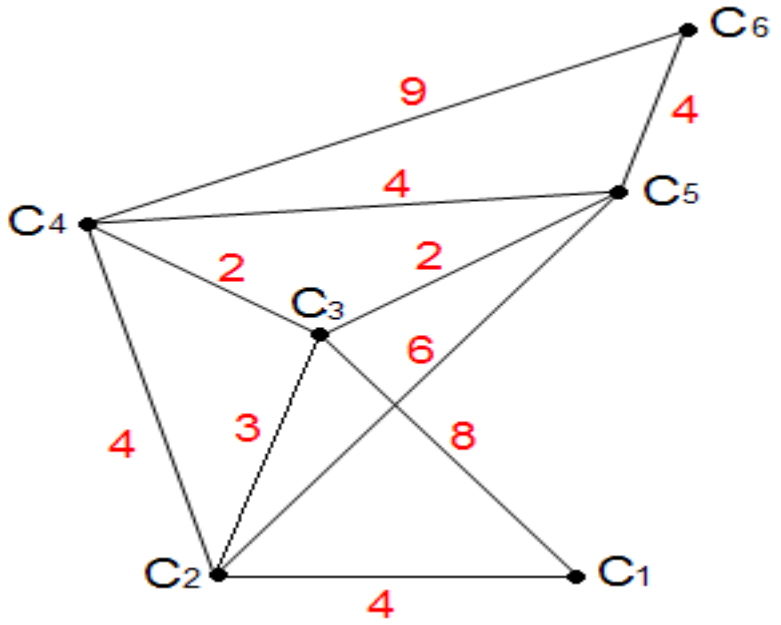
```
Initalize-Single-Source(G,s)
1 for each vertex v∈V[G]
2 do d[v]←∞
3 pre[v]←Null
4 d[s]←0
```

Dijkstra算法

- Dijkstra算法是求单源最短路算法，采用的是贪心的策略，适用所有边的权值非负。Dijkstra算法中设置了一结点集合T，从源结点s到集合T中结点的最终最短路径的权均已确定，即对所有结点v∈T，有d[v]=(s,v)的权值。算法反复挑选出其最短路径估计为最小的结点u∈V-T，把u插入集合T中，并对离开u的所有边进行松弛。

```
1.  INITIALIZE-SINGLE-SOURCE(G,S)
2.  Q ← V[G]
3.  While Q
4.      { u ← EXTRACT-MIN(Q)
4.       T ← T U {u}
6.       For 每个顶点 v∈Adj[u]   RELAX(u,v,w) }
```

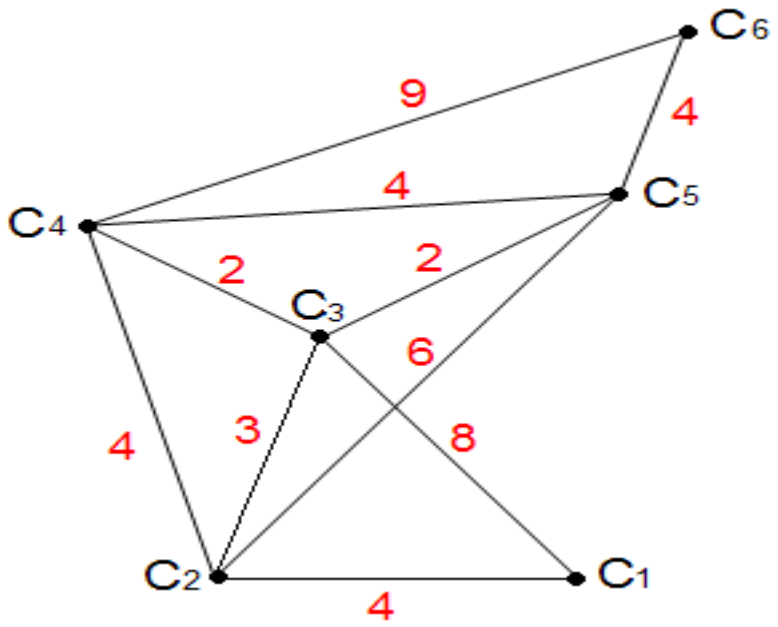
Dijkstra算法



初始时：$T=[C_1]$

i	1	2	3	4	5	6
D	0	4	8	∞	∞	∞
P	C_1	C_1,C_2	C_1,C_3			

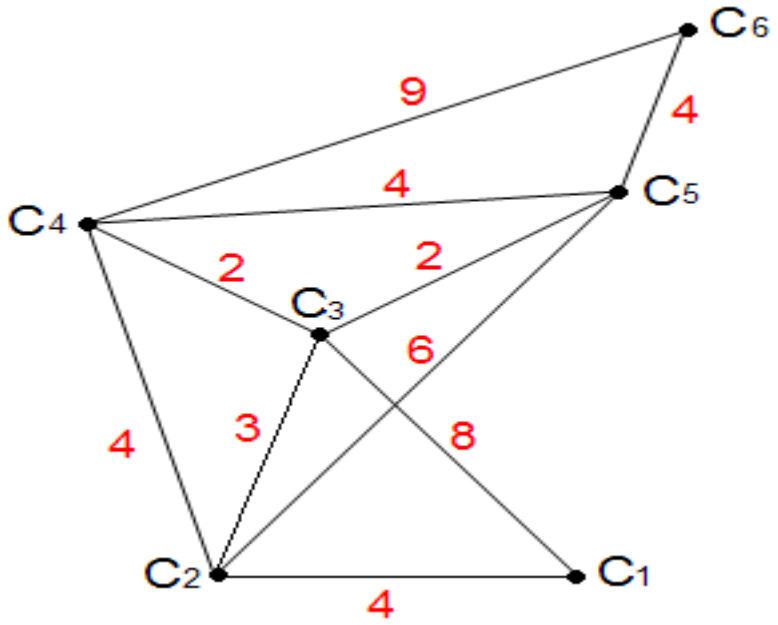
Dijkstra算法



第一次:选择i=2,T=[C_1,C_2],计算比较D[2]+G[2,j]与D[j]的大小(3≤j≤6)

i	1	2	3	4	5	6
D	0	4	8->7	∞->8	∞->10	∞
P	C_1	C_1,C_2	C_1,C_2,C_3	C_1,C_2,C_4	C_1,C_2,C_5	

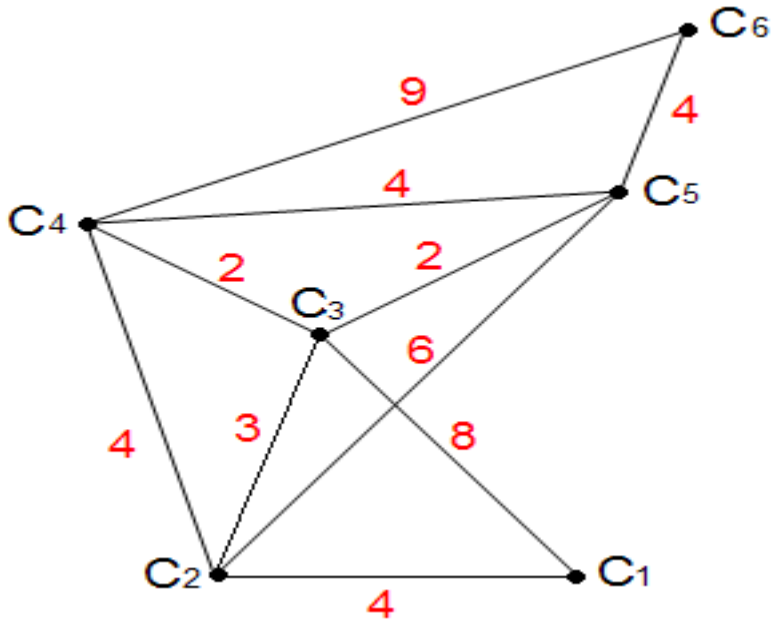
Dijkstra算法



第二次:选择i=3,T=[C_1,C_2,C_3],计算比较D[3]+G[3,j]与D[j]的大小(4≤j≤6)

i	1	2	3	4	5	6
D	0	4	7	8	10->9	∞
P	C_1	C_1,C_2	C_1,C_2,C_3	C_1,C_2,C_4	C_1,C_2,C_3,C_5	

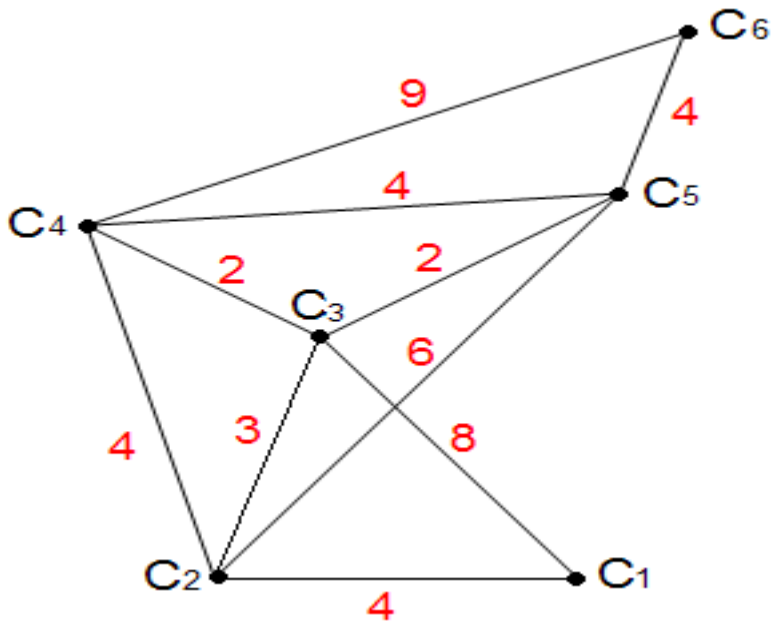
Dijkstra算法



第三次:选择i=4,T=[C_1,C_2,C_3,C_4],计算比较D[4]+G[4,j]与D[j]的大小(5≤j≤6)

i	1	2	3	4	5	6
D	0	4	7	8	9	∞->17
P	C_1	C_1,C_2	C_1,C_2,C_3	C_1,C_2,C_4	C_1,C_2,C_3,C_5	C_1,C_2,C_4,C_6

Dijkstra算法



第四次:选择i=5,T=[C_1,C_2,C_3,C_4,C_5],计算比较D[5]+G[5,j]与D[j]的大小(j=6)

i	1	2	3	4	5	6
D	0	4	7	8	9	17->13
P	C_1	C_1,C_2	C_1,C_2,C_3	C_1,C_2,C_4	C_1,C_2,C_3,C_5	C_1,C_2,C_3,C_5,C_6

Dijkstra算法

```
int dijkstra(int s) //求源点s到各个点的最短路
{for(int u=1; u<=n; u++)  d[u] = oo;  //初始化
 d[s] = 0;
 for (int i=1; i<=n; i++) //需要把n个点的最短路确定
   {int min = oo, u = 0;
    for (int v = 1; v <= n; v++) //查找不在T集合且d[ ]最小的
      if (!flag[v] && d[v] < min)  min = d[v], u = v;
    flag[u] = 1; //标记点u的最短路确定,即加入T集合
    for (int v = 1; v <= n; v++) //对离开u的所有边进行松弛
       if (!flag[v] && d[u] +w[u][v] < d[v])  d[v] = d[u] + w[u][v];
   }
}
```

Dijkstra算法的优化

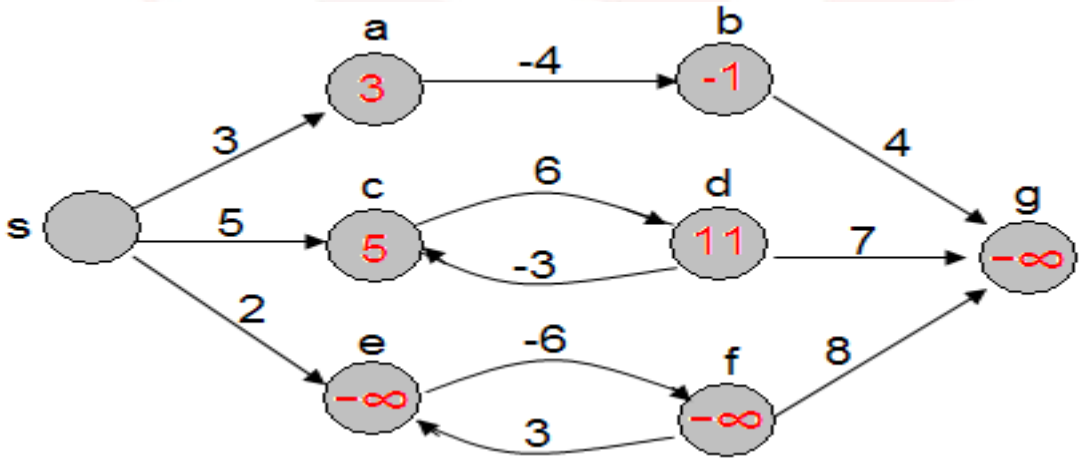
- 前面介绍的Dijkstra算法的时间复杂度为$O(n^2)$。
- Dijkstra算法的优化主要基于以下两个方面：
  - 用邻接表代替邻接矩阵；
  - **堆优化：将当前不在T集合中的所有结点的d[ ]放到一个小根堆中，这样就可以O(1)查找不在T集合且d[ ]最小的结点u，从堆中删除u，并对离开u的所有边进行松弛，更新在堆中的d[ ]，手写堆时利用pos[i]记录顶点i在堆中的位置，时间复杂度为O((n+e)logn)。利用STL中的堆，时间复杂度为O((n+e)loge)**

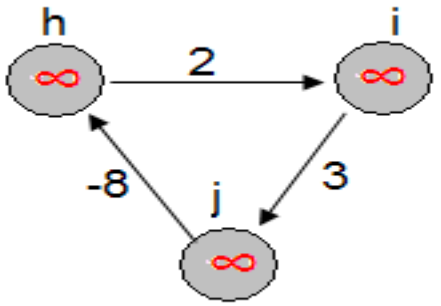
```
void Dijkstra(int s) //写手堆优化Dijkstra
{for (int i=1; i<=n; i++) d[i]=oo;  d[s]=0; //初始化
 tot=0; //堆中结点个数
 for (int i=1; i<=n; i++) Insert(i); //将n个顶点的(adj和dist)加入堆
 while (1)
   {HeapTp zx=Min(); //取出堆中当前dist最小的结点
    d[zx.adj]=zx.dist;  //源点s到这个顶点的最短路已确定
    if (tot==0) break;  //堆中无顶点,即n个顶点的最短路都已求出
    for (int i=0; i<g[u].size(); i++) //将它相关联所有顶点的dist更新
     {int k=pos[g[u][i].adj]; //关联顶点在堆中的位置
      if (k<=tot && zx.dist+g[u][i].dist<heap[k].dist) //松弛
       {heap[k].dist=zx.dist+g[u][i].dist;
        Heap_Up(k); //往上调整使它满足小根堆
       }
}}}
```

```
priority_queue<int,vector<int>,greater<int>> Q; //小根堆
void Dijkstra(int s) //STL堆优化Dijkstra
{for (int i=1; i<=n; i++) {flag[i]=false; d[i]=oo; } //初始化
 d[s]=0; Q.push(s);
 while (!Q.empty())  //队列不为空
   {int u=Q.top(); Q.pop();
    if (flag[u]) continue;  //d[u]已经确定
    flag[u]=true;
    for (int i=head[u];  i;  i=Next[i])  //处理u的所有关联顶点v
      {int v=adj[i];
       if (!flag[v] && d[v]>d[u]+val[i])
         {d[v]=d[u]+val[i]; Q.push(v); }
      }
   }
}
```

SSSP：权任意的情形

- 最短路有可能不存在！
  - 什么时候不存在？
  - 有负权回路！





Bellman-Ford算法

- Bellman-Ford算法运用了松弛技术，对每一结点v∈V，逐步减小从源s到v的最短路径的估计值d[v]直至其达到实际最短路径的权δ(s,v)，如果图中存在负权回路，算法将会报告最短路不存在。

```
Bellman-Ford(G,w,s)
1.  INITIALIZE-SINGLE-SOURCE(G,s)
2.  For i ← 1 to |V[G]|-1
3.      Do  For 每条边(u,v) ∈ E[G]
4.          Do  RELAX(u,v,w)
5.  For 每条边(u,v)∈ E[G]
6.      Do  If d[v] > d[u] + w(u, v)
7.          Then Return FALSE
8.  Return TRUE
```

Bellman-Ford算法

- 适用条件：任意边权为实数的图。
- Bellman-Ford算法的思想基于以下事实：两点间如果有最短路，那么每个结点最多经过一次。也就是说，这条路不超过n-1条边。（如果一个结点经过了两次，那么我们走了一个圈。如果这个圈的权为正，显然不划算；如果是负圈，那么最短路不存在；如果是零圈，去掉不影响最优值）
- Bellman-Ford算法的运行时间为O(VE)。很多时候，我们的算法并不需要运行|V|-1次就能得到最优值。对于一次完整的第3-4行操作，要是一个结点的最短路径估计值也没能更新，就可以退出了。

Bellman-Ford算法

- 算法流程分三个阶段：
  - 初始化：将除源点s外的所有顶点的最短距离估计值d[v]=∞,d[s]=0;
  - 迭代求解：反复对边集E中的每条边进行松弛操作，使从源s到v的最短路径的估计值d[v]逐步逼近实际最短路径的权δ(s,v)

```
for (k=1;k<=n-1;k++)
  for 每条边(u,v)
    if ((d[u]<∞ && d[v]>d[u]+w(u,v))  d[v]=d[u]+w(u,v)
```

  - 验收负权回路：判断边集E中的每一条边的两个端点是否收敛。

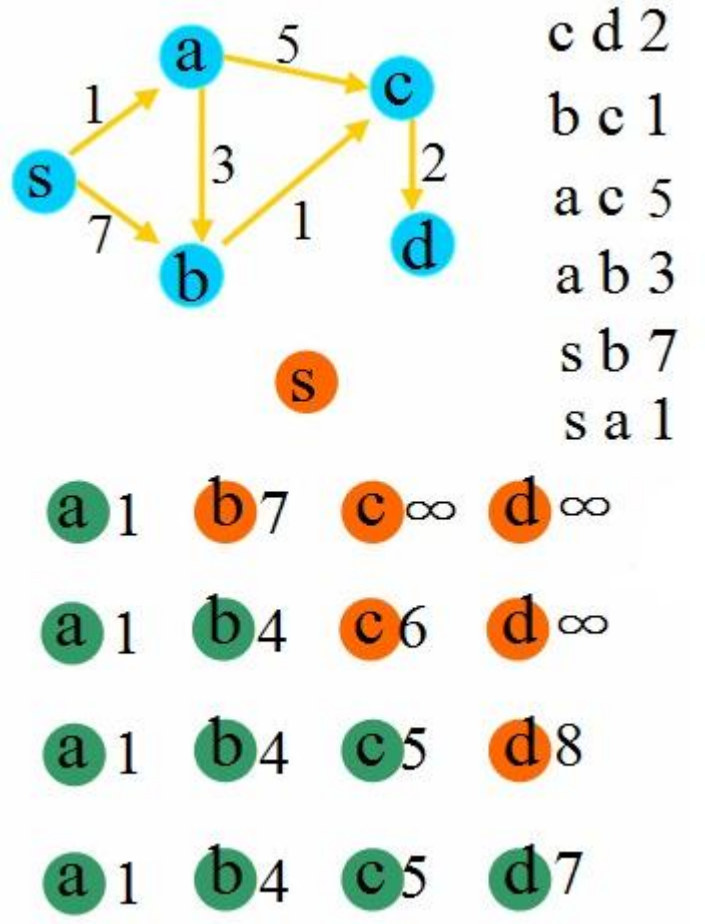
程序

```
bool bellman_ford(int s)
{for (int i=1; i<=n; i++) d[i]=oo;   d[s]=0; //初始化
 for (int i=1; i<=n-1; i++)  //最多进行n-1次迭代
   {bool flag=false;
    for (int j=1; j<=m; j++) //边集E中的每条边进行松弛操作
      if (d[e[j].u] < oo && d[e[j].v] > d[e[j].u]+e[j].w)
        {d[e[j].v]=d[e[j].u]+e[j].w; flag=true;}
    if (!flag) break;  //最短路径估计值没更新
   }
 for (int j=1; j<=m; j++)
   if (d[e[j].v] > d[e[j].u]+e[j].w) return false; //有负权回路
 return true;
}
```

SPFA算法

- SPFA算法简单地说就是队列优化的Bellman-Ford算法。
- Bellman-Ford算法的迭代松弛操作，最坏情况是按顶点距离源点s的层次，逐层生成这棵最短路径树的过程。
- 对每条边进行第1遍松弛的时候，生成了从s出发层次为1的那些树枝。也就是说，找到了与s至多有1条边相联的那些顶点的最短路径；对每条边进行第2遍松弛的时候，生成了第2层次的树枝，就是说找到了经过2条边相连的那些顶点的最短路径……。因为最短路径最多只包含|v|-1条边，所以需要循环|v|-1 次。

SPFA算法



- 如左图，每次迭代时6条边按从上往下的顺序进行松弛，需要进行四次迭代，每次迭代都需要松弛6条边，分别得到源点s经过1条边、2条边、3条边、4条边的最短路径。
- 但实际上，第一次松弛边(s,a)和(s,b)，第二次松弛边(a,b)、(a,c)和(b,c)，第三次松弛(c,d)就可以了。

SPFA算法

- 每实施一次松弛操作，最短路径树上就会有一层顶点达到其最短距离，此后这层顶点的最短距离值就会一直保持不变，不再受后续松弛操作的影响。但是，每次还要判断松弛，这里浪费了大量的时间，这就是Bellman-Ford算法效率低下的原因，也正是SPFA优化的所在。
- 只有那些在前一遍松弛中改变了距离估计值的点，才可能引起他们的邻接点的距离估计值的改变。
- 时间复杂度：O(KE)，K是常数，E是边数。比Bellman-Ford算法效率高。

SPFA算法

- 我们用数组d记录每个结点的最短路径估计值，而且用邻接表来存储图G。我们采取的方法是动态逼近法。
- 设立一个先进先出的队列用来保存待优化的结点，优化时每次取出队首结点u，并且用u点当前的最短路径估计值对离开u点所指向的结点v进行松弛操作，如果v点的最短路径估计值有所调整，且v点不在当前的队列中，就将v点放入队尾。这样不断从队列中取出结点来进行松弛操作，直至队列空为止。
- SPFA 在形式上和宽度优先搜索非常类似，不同的是宽度优先搜索中一个点出了队列就不可能重新进入队列，但是SPFA中一个点可能在出队列之后再次被放入队列。

SPFA算法

```
void spfa(int s)
{int first=0,tail=1; //队首指针,队尾指针
 for (int i=1; i<=n; i++) d[i]=oo;
 d[s]=0; q[tail]=s; flag[s]=true; //初始化,源点s进队列并标记s在队列中
 while (first<tail) //队列不为空
   {int u=q[++first]; flag[u]=false; //队首结点u出队列
    for(i=g[u]; i>0; i=e[i].next) //处理和u相关的每一条边
      {int v=e[i].v, w=e[i].weight;
       if (d[v]>d[u]+w) //松弛
        {d[v]=d[u]+w;
         if (!flag[v]) {q[++tail]=v;flag[v]=true;} //不在队列中则进队列
} } } }
```

SPFA算法

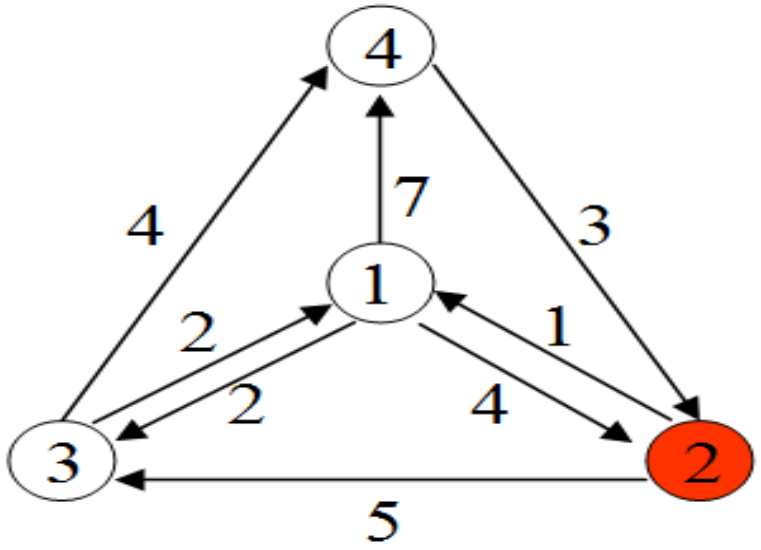
- SPFA无法处理带负环的图，怎样判断图中有负权回路：
  - 如果某个点进入队列的次数超过N次则存在负环，假设这个节点的入度是k(无向权则就是这个节点的连接的边)如果进入这个队列超过k，说明必然有某个边重复了，即成环;
  - 用DFS，假设存在负环a1→a2→…→an→a1。那么当从a1深搜下去时又遇到了a1，那么直接可以判断负环了。

SPFA算法

- SPFA的优化方法：
  - 循环队列（可以降低队列大小）
  - **SLF：Small Label First 策略，设要加入的节点是j，队首元素为i，若d[j] < d[i]，则将j插入队首，否则插入队尾。**
  - **LLL：Large Label Last 策略，设队首元素为i，每次弹出时进行判断，队列中所有dist值的平均值为x，若d[i]>x则将i插入到队尾，查找下一元素，直到找到某一i使得d[i]<=x，则将i出对进行松弛操作。这个优化因时间复杂度较高一般不使用。**

例题：派对

- N头牛要去参加一场在编号为 x(1≤x≤n)的牛的农场举行的派对(1≤N≤1000)，有M(1≤M≤100000)条有向道路，每条路长t[i](1≤t[i]≤100)。每头牛都必须参加完派对后回到家，每头牛都会选择最短路径，求这 n 个牛的最短路径（一个来回）中最长的一条的长度。
- 特别提醒：可能有权值不同的重边。



分析

- 本题求所有奶牛一个来回的最短距离的最大值。
- 每头奶牛从编号为x的结点回去的最短路，就是把x当源点，求x到各个点的最短路，这是单源最短路问题，设D1[i]表示x到i的最短路
- 每头奶牛来到编号为x的结点的最短路稍微复杂点，相当于在反图上（因为是有向图）求从x出发到各个点的最短路，设D2[i]表示x到i的最短路，也就是i到x的最短路。
- ans=max{D1[i]+D2[i]}，可以用Dijkstra或SPFA。

例题：非常计划

- Smart 开启了时光隧道来到了公元前 30 世纪，准备前往埃及。在他的地图上，有N个城市，我们已知他目前处在城市 1，目的地在城市N，每一条道路都是单向的。我们还知道，从i城市到j城市需要D[i][j]的花费。
- Smart 想走一条从城市1到城市N花费最少的一条路，Smart 是一个喜欢思考的小盆友，他还希望你能告诉他花费最少的路径共有多少条(两个不同的最短路方案要求：路径长度相同（均为最短路长度）且至少有一条边不重合)。
- $1 \le N \le 2000$，$0 \le E \le N(N-1)$。

分析

- 因为1≤N≤2000，0≤E≤N(N-1)，因为图为稠密图，所以采用Dijkstra算法求最短路。
- 设dis[i]表示1号点到i号点的最短路，sum[i]表示1号点到i号点的最短路的路径条数，初始化：dis[1]=0,sum[1]=1。
- 每次找一个不在T集合中且dis[ ]最小的结点u，来更新不在T集合中的其他结点v，若:
  - dis[v]>dis[u]+D[u][v],则dis[v]=dis[u]+D[u][v],sum[v]=sum[u];
  - dis[v]=dis[u]+D[u][v],则sum[v]+=sum[u];
- 最后最短路径为dis[N]，最短路径条数为sum[N]。

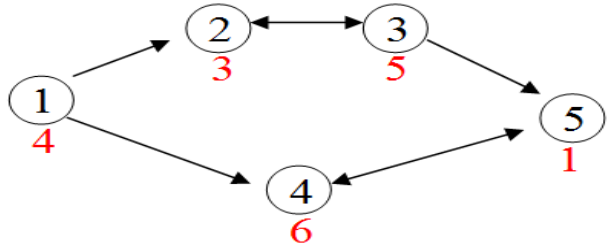
中国计算机学会
China Computer Federation

例题：最优贸易

- C国有n个大城市和m条道路，每条道路连接这n个城市中的某两个城市。任意两个城市之间最多只有一条道路直接相连。这m条道路中有一部分为单向通行的道路，一部分为双向通行的道路，双向通行的道路在统计条数时也计为1条。
- C国幅员辽阔，各地的资源分布情况各不相同，这就导致了同一种商品在不同城市的价格不一定相同。但是，同一种商品在同一个城市的买入价和卖出价始终是相同的。
- 商人阿龙来到C国旅游。当他得知同一种商品在不同城市的价格可能会不同这一信息之后，便决定在旅游的同时，利用商品在不同城市中的差价赚回一点旅费。设C国n个城市的标号从1~ n，阿龙决定从1号城市出发，并最终在n号城市结束自己的旅行。$1 \le n \le 100000$，$1 \le m \le 500000$

例题：最优贸易

- 在旅游的过程中，任何城市可以重复经过多次，但不要求经过所有n个城市。阿龙通过这样的贸易方式赚取旅费：他会选择一个经过的城市买入他最喜欢的商品——水晶球，并在之后经过的另一个城市卖出这个水晶球，用赚取的差价当做旅费。由于阿龙主要是来C国旅游，他决定这个贸易只进行最多一次，当然，在赚不到差价的情况下他就无需进行贸易。
- 现在给出 n个城市的水晶球价格，m条道路的信息（每条道路所连接的两个城市的编号以及该条道路的通行情况）。请你告诉阿龙，他最多能赚取多少旅费。



分析

- 设minp[i]表示从1到i号城市能买到水晶球的最低价格，maxf[i]从1到i号城市能卖出水晶球的最高价格。
- 初始化：
  - minp[i]=∞，maxf[i]=0（1≤i≤n）
  - minp[1]=price[1] (price[i]表示城市i水晶球的价格)。
- 利用SPFA的思想维护，对于一条边<u,v>松弛时：
  - 需满足minp[v]>minp[u] 或maxf[v]<maxf[u]
  - 更新minp[v]=min(price[v], minp[u]);
  - 更新maxf[v]=max(maxf[u], price[v]-minp[v]);

中国计算机学会
China Computer Federation

例题：电话网络

- 在郊区有N座通信基站，P条双向电缆，第i条电缆连接基站Ai和Bi。特别地，1号基站是通信公司的总站，N号基站位于一座农场中。现在农场主希望对通信线路进行升级，其中，升级第i条电缆需要花费Li。
- 电话公司正在举行优惠活动。农场主可以指定一条从1号基站到N号基站的线路，并指定线路上不超过K条电缆，由电话公司免费提供升级服务。农场主只需要支付在该线路上剩余的电缆中，升级价格最贵的那条电缆的费用即可。求至少用多少钱才能完成升级。
- $0 \le K < N \le 1000, 1 \le P \le 10000$。

分析

- 简单地说，本题是在无向图上求一条由1到N的线路，使得线路上第K+1大的边权尽量小。
- 本题的结果显然具有单调性，所以可以二分答案。首先二分路径中第K+1大的边权长度mid，判定问题非常容易，有一个很好的简化方法，就是把升级价格大于mid的电缆长度置为1，代表要使用一次免费机会，把升级价格不超过mid的电缆长度置为0，这样再跑SPFA，看1到N的最短路长度len是否不超过K即可。若len>K，则要增大mid，反之可以缩小mid。

例题：序列长度

- 有一个序列，题目用n个整数组合$[a_i,b_i,c_i]$来描述它，$[a_i,b_i,c_i]$表示在该序列中处于区间$[a_i,b_i]$内的整数至少有c_i个。如果存在这样的序列，请求出满足题目要求的最短的序列长度是多少。如果不存在则输出-1。
- 第一行包括一个整数$n(n\leq1000)$，表示区间的个数，以下n行每行描述这些区间，第i+1行三个整数a_i,b_i,c_i，由空格隔开，其中$0\leq a_i\leq b_i\leq1000$且$1\leq c_i\leq b_i-a_i+1$。

中国计算机学会
China Computer Federation

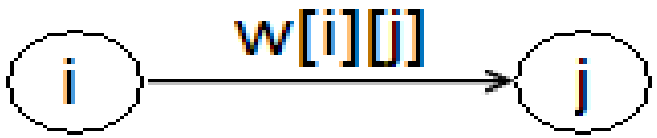
例题：序列长度

- 输出一行，输出满足要求的序列的长度的最小值。
- 样例：

interval.in	interval.out
5 3 7 3 8 10 3 6 8 1 1 3 1 10 11 1	6

分析

- 这道题是一道差分约束系统的经典例题。
- 有向图中“最短路径”：设d[i]表示源点s到顶点i的最短路径，那么对于任意一条从顶点i射出去的有向边<i,j>，如下图：



- 能得到如下结论：
  - **d[i]+w[i][j]≥d[j]→d[j]-d[i]≤w[i][j];**
  - 有负权回路的有向图不存在最短路径

分析

- 设x[i]表示在区间[0,i]内至少有多少个整数
- x显然应该满足的条件：
  - (1) **x[i]-x[i-1]≥0**
  - (2)**x[i]-x[i-1]≤1→x[i-1]-x[i]≥-1**
- 根据题目描述中提出的$[a_i,b_i,c_i]$表示在该序列处于区间$[a_i,b_i]$内的整数至少有c_i个，那么可以得到如下不等式：$x[b_i]-x[a_i-1]\geq c_i$

分析

- 根据条件建立有向图：
  - 将x[0],x[1],…,x[n]看作是n+1个点$v_0,v_1,\ldots,v_n$;
  - 若x[i]-x[j]之间有约束关系，如x[i-1]-x[i]≥-1,那么就从结点v_{i-1}向结点v_i连上一条有向边，边的权值为-1;
- 方法：从v_0出发，求出结点v_0到v_n的最短路径，则x[n]为满足要求情况下的最小值，如果有负权回路，说明不存在。

拓扑排序算法

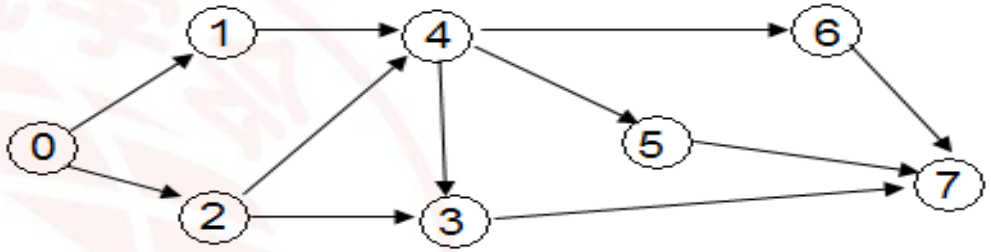
- 在日常生活中，一项大的工程可以看作是由若干个子工程（这些子工程称为“活动”）必须在其他一些子工程（活动）完成之后才能开始，我们可以用有向图来形象地表示这些子工程（活动）之间的先后关系，子工程（活动）为顶点，子工程（活动）之间的先后关系为有向边，这种有向图称为“顶点活动网络”，又称“AOV网”。在AOV网中，有向边代表子工程（活动）的先后关系，即有向边的起点活动是终点活动的前驱活动，只有当起点活动完成后终点活动才能够进行。

拓扑排序算法

- 如果有一条从顶点v_i到v_j的路径，则说v_i是v_j的前驱，v_j是v_i的后继。如果有弧$<v_i,v_j>$，则称v_i是v_j的直接前驱，v_j是v_i的直接后继。
- 一个AOV网应该是一个有向无环图，即不应该带有回路，否则必定会有一些活动互相牵制，造成环中的活动都无法进行。
- 把不带回路的AOV网中的所有活动排成一个线性序列，使得每个活动的所有前驱活动都排在该活动的前面，这个过程称为“拓扑排序”。

拓扑排序算法

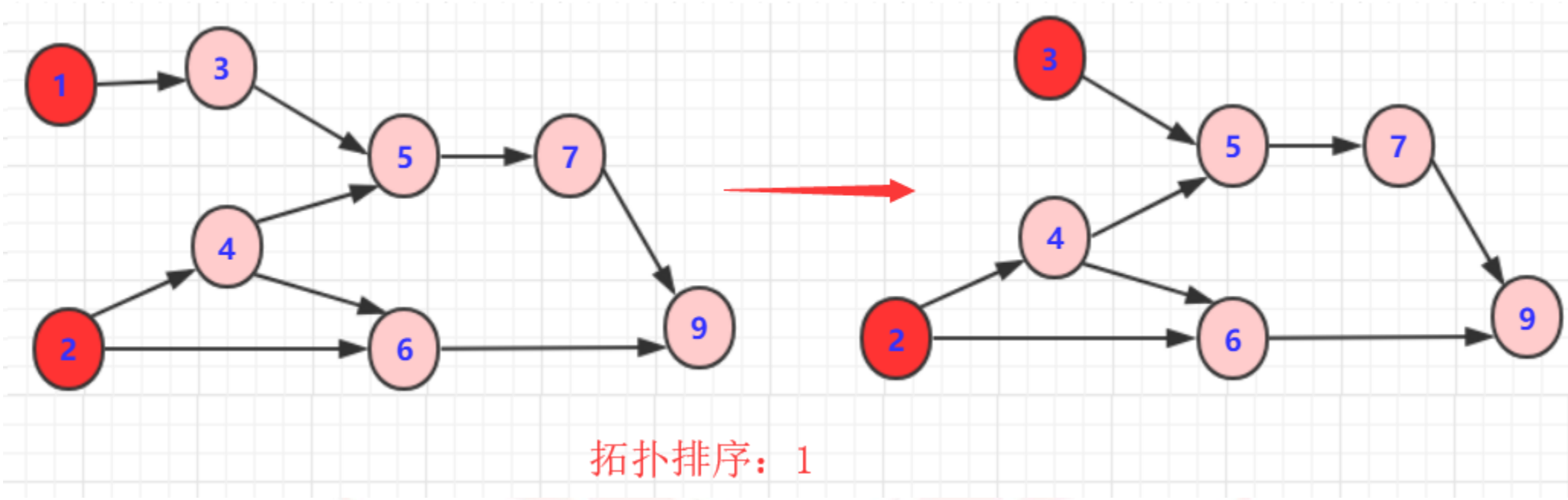
- 如图，进行拓扑排序至少有如下几种拓扑序列：02143567，01243657，02143657，01243567。



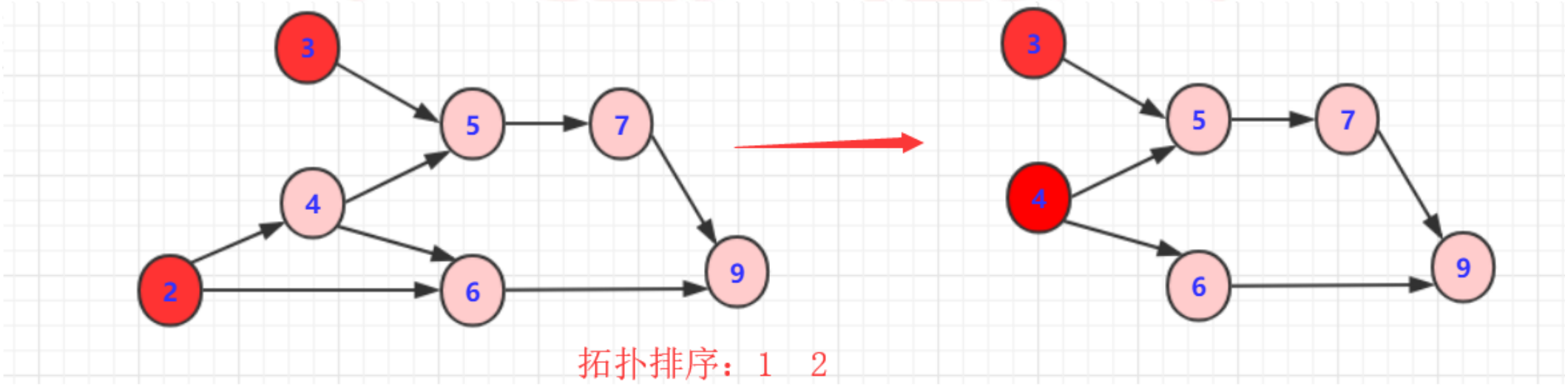
- 拓扑排序算法：
  - 只要选择一个入度为0的顶点并输出，
  - 然后从AOV网中删除此点以及此顶点为起点的所有关联边;
  - 重复以上两步，直到不存在入度为0的顶点，若输出的顶点数小于AOV网中的顶点数，则输出“有回路信息”，否则输出的就是一种拓扑排序。

中国计算机学会
China Computer Federation

求拓扑序列

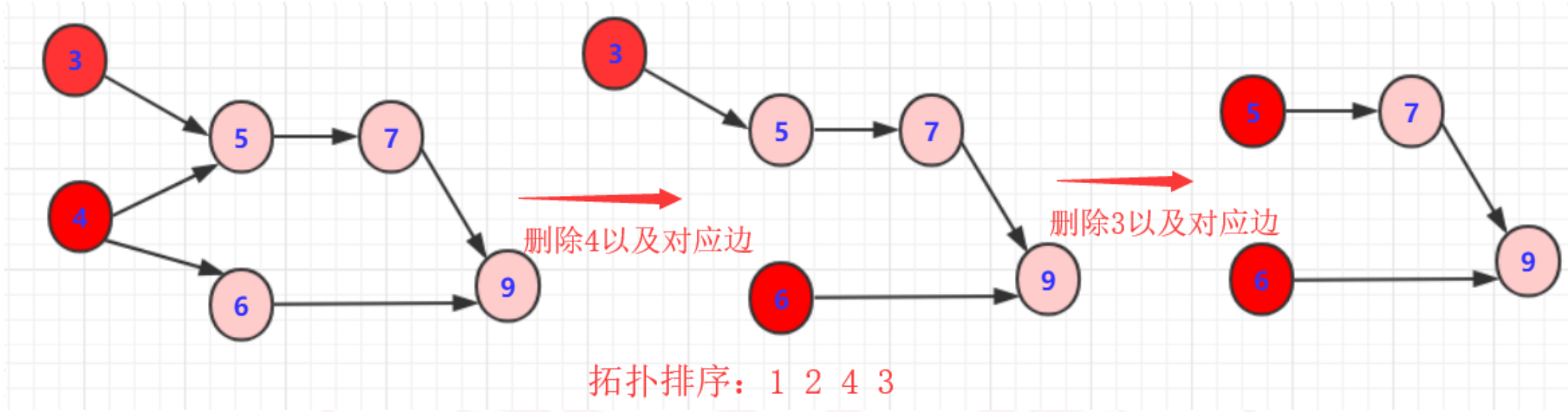


拓扑排序：1

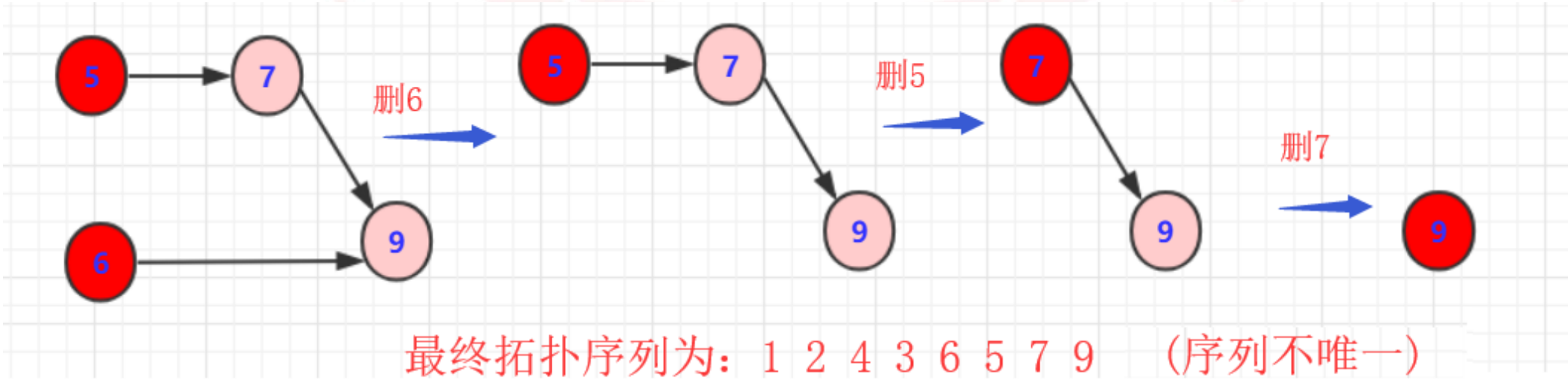


拓扑排序：1　2

求拓扑序列



拓扑排序：1 2 4 3



最终拓扑序列为：1 2 4 3 6 5 7 9 （序列不唯一）

例题：复杂的按钮

- 小Y在遗迹探险时遇到了n个按钮，刚开始所有按钮都处于开状态，小Y的经验告诉他把所有的按钮都关上会有“好事”发生，可是有些按钮按下时会让其他一些已经闭合的按钮弹开。经过研究，每个按钮都对应着一个固定的弹开集合，这个按钮按下时，弹开集合中所有的按钮都会变成开状态。现在小Y想知道是否能让所有的按钮变为闭状态。如果能，打印最少的步数以及方案，否则，打印"no solution"。
- 100%的数据：$1 \le n \le 30000$, $0 \le m_1+m_2+\ldots+m_n \le 1000000$。

例题：复杂的按钮

- 第一行一个整数n，表示有n个按钮
  接下来n行，表示编号为1到n个按钮的弹开按钮集合格式为m_i，表示编号为i的按钮按下，会让编号为$B1,B2,...,Bm$的按钮弹开（注：其中不会出现重复）
- 如果无解，输出"no solution"。否则，第一行输出最少步数ans，第二行输出用空格分开的ans个数，表示按顺序按下编号这些数的按钮就可以解决。如果有多种方案，输出字典序最小的方案。

倍增查找LCA

- LCA（Least Common Ancestors），即公共最近祖先，是指在有根树中找出某两个节点u和v最近的公共祖先。

倍增查找LCA

- 方法一
  - 暴力找
  - 首先将u和v中深度较深的那个点蹦到和深度较浅的点同样的深度。然后两个点一起向上蹦，直到蹦到同一个点，这个点就是它们的LCA。
  - 复杂度：极限情况可以达到O(n)。

倍增查找LCA

- 方法二
  - 利用DFS序
  - DFS序就是用DFS方法遍历整棵树得到的序列。
  - 两个点的LCA一定是两个点在DFS序中出现的位置之间深度最小的那个点。
  - 如何寻找最小值？
  - 使用刚刚学习的RMQ。

倍增查找LCA

- 方法三
  - 设father[i][j]表示编号为i的点，往上蹦2^j的祖先是谁
  - 这个预处理与RMQ的预处理类似。先从小到大枚举j，然后令
  - father[i][j]=father[father[i][j-1]][j-1]。
  - 预处理的时间复杂度为O(nlogn)

倍增查找LCA

- 方法三
  - 那如何查询呢？
  - 分两步走
    - ✓ 将u和v移动同样的深度
    - ✓ u和v同时向上移动，直到重合。第一个重合的点即为LCA

倍增查找LCA

- 方法三
  - 移动同样的深度
    - ✓ 令u为深度较大的点。我们从$\log_2 n$到0枚举，令枚举的数字为j。如果从u向上蹦2^j步小于了v的深度，不动；否则向上跳2^j步，这样一定能移动到和v同样的深度。
    - ✓ 假设一共要蹦k步，上面的算法相当于是枚举k的每个二进制位是0还是1。

倍增查找LCA

- 方法三
  - 从同样的深度移动到同一个点
    - ✓ 和上一步类似。从$\log_2 n$到0枚举，设枚举的数字为j。如果两个点向上蹦2^j步将要重合，不动；否则向上蹦2^j步。
    - ✓ 通过这种方法，u和v一定能够到达这样一个状态——他们当前不重合，如果再往上蹦一步，就会重合。所以再往上蹦一步得到的就是LCA。
    - ✓ 因为本质上是枚举每个二进制位是0还是1，所以单次查询的时间复杂度为$\log_2 n$。

Thanks