作业回顾

1.矩阵运算

$$\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}\begin{pmatrix}5 & 6\\ 7 & 8\end{pmatrix}=\begin{pmatrix}19 & 22\\ 43 & 50\end{pmatrix}\qquad \begin{pmatrix}1 & 2 & 3 & 4\end{pmatrix}\begin{pmatrix}5\\ 6\\ 7\\ 8\end{pmatrix}=\begin{pmatrix}70\end{pmatrix}$$

矢量内积

$$\begin{pmatrix}1\\ 2\\ 3\\ 4\end{pmatrix}\begin{pmatrix}5 & 6 & 7 & 8\end{pmatrix}=\begin{pmatrix}5 & 6 & 7 & 8\\ 10 & 12 & 14 & 16\\ 15 & 18 & 21 & 24\\ 20 & 24 & 28 & 32\end{pmatrix}$$

线性相关

2.矩阵乘法结合律

$$\begin{aligned}((A_{m*n}*B_{n*p})*C_{p*q})_{ij} &= \sum_{k=1}^{p}((AB)_{ik}*C_{kj})\\
&=\sum_{k=1}^{p}((\sum_{l=1}^{n}A_{il}*B_{lk})*C_{kj})=\sum_{k=1}^{p}(\sum_{l=1}^{n}A_{il}*B_{lk}*C_{kj})\\
&=\sum_{l=1}^{n}(\sum_{k=1}^{p}A_{il}*B_{lk}*C_{kj})=\sum_{l=1}^{n}(A_{il}*(\sum_{k=1}^{p}B_{lk}*C_{kj}))\\
&=\sum_{l=1}^{n}(A_{il}*(BC)_{lj})=(A(BC))_{ij}\end{aligned}$$

3.随机数列

→ **明显是线性变换，要凑成矩阵形式**

$$\begin{pmatrix} X_{n+1} \\ 1 \end{pmatrix} = \begin{pmatrix} a & c \\ 0 & 1 \end{pmatrix} \begin{pmatrix} X_n \\ 1 \end{pmatrix}$$

→ **常数矩阵可以用快速幂**

这个矩阵有一些特殊性，你会发现它的**任意次幂第二行总是（0，1）**

所以可以简化下，只记录第一行的两个元素就可以了

→ **还有一个细节问题**

由于元素大小可以到10^18，如果直接相乘会出现中间结果越界

因此可以用二进制的方式逐位相乘并取模（参见代码）

如果你愿意也可以用高精度，不过有点浪费

→ **复杂度：O(logn)**

3.随机数列

```
LL mm(LL x, LL y){  // 防越界的LL乘法 (mod m)
    if (!y) return 0;
    LL ans = mm(x, y >> 1) << 1;
    if (y & 1) ans += x;
    return ans % m;
}

inline matrix mul(matrix a, matrix b){  // 矩阵乘法 (特殊版)
    return (matrix){mm(b.a,a.a),(mm(b.a,a.c)+b.c)%m};
}


matrix pow(matrix a, LL t){ // 矩阵幂次
    if (t==0) return id;
    matrix p = pow(a, t >> 1);
    p = mul(p, p);
    return t&1 ? mul(a, p) : p;
}

int main(){
    scanf("%lld%lld%lld%lld%lld%lld", &m,&a,&c,&x,&n,&g);
    matrix s = (matrix){a,1};
    matrix p = pow(s, n);    // 幂次
    printf("%lld", (mm(p.a, x) + mm(p.c, c))%m%g);
    return 0;
}
```

4.传球问题

→ **不考虑性能问题，这题应该是动规（之前CS200做到过类似的题）**

阶段：传几次，O(k)

状态：从谁开始传到谁，O(n2)

决策：上一个传过来的人，O(n)

$$f_{ij}(k) = \sum_{\substack{x=1 \\ a_{xj}=1}}^{n} f_{ix}(k-1) = \sum_{x=1}^{n} f_{ix}(k-1)a_{xj}$$

→ **不看k指标，不就是个矩阵乘法吗…**

F(k)=F(k-1)A=…=A^k

```
int n,k;
int main(int argc, const char * argv[]) {
    cin>>n>>k;
    LL** mt=newMatrix(n);
    for (int i=0;i<n;i++){
        for (int j=0;j<n;j++) cin>>mt[i][j];
    }
    LL** ans=pow(mt,n,k);
    print(ans,n);
    return 0;
}
```

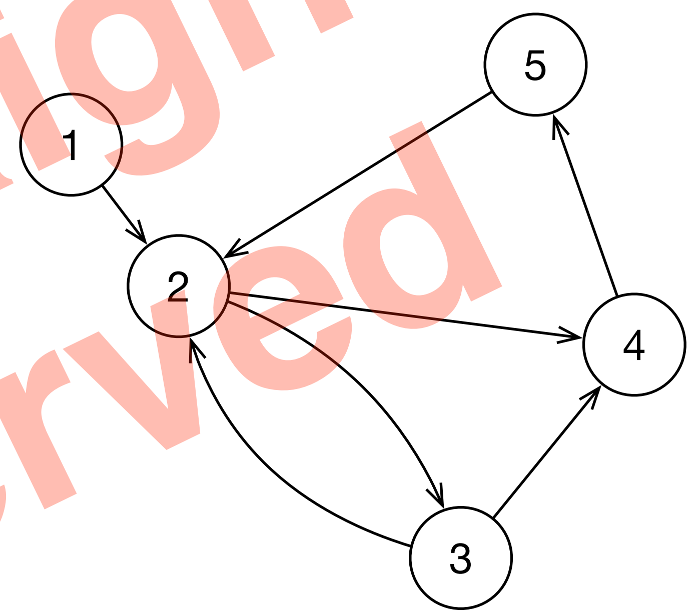
→ **复杂度O(n3logk)**

→ **注意小心中间结果越界**

矩阵乘法的意义：分步计数

→ **A称为图的相邻矩阵（马上就会讲到）**

$$\begin{matrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{matrix}$$



→ **为什么路径计数正好等于矩阵乘法，巧合?**

乘法原理：a→b→c的路径数=a→b的路径数*b→c的路径数

加法原理：a→c的路径数=Σa→b→c的路径数（对所有b求和）

以上正是矩阵乘法的定义（先乘后加）

5.旋转矩阵

→ **这是一道混淆题~其实跟矩阵并没太大关系😏**

如果你傻呵呵地搞一个二维矩阵去模拟，那么时空复杂度都是O(n^2)

→ **其实稍微用数学算算就可以了**

```
int main(int argc, const char * argv[]) {
    cin>>n>>x>>y;
    long long outer=min(min(x,y),min(n-x+1,n-y+1))-1;   // 外层圈数
    long long ans=n*n-(n-outer*2)*(n-outer*2);  // 外圈总数
    x-=outer,y-=outer,n-=outer*2;  // 移除外圈
    if (x==1) ans+=y;  // 上边
        else if (y==n) ans+=n+x-1;    // 右边
        else if (x==n) ans+=2*n-1+(n-y);   // 下边
        else ans+=3*n-2+(n-x);   // 左边
    cout<<ans<<endl;
    return 0;
}
```

→ **复杂度O(1)**

→ **如果你懒一点，一条条边去模拟，O(n)也是可以的**

CS102 算法提高

最短路

先讲一点科学：黑洞

→ **说到黑洞，不得不提高大上的广义相对论**

广义相对论是时空结构和引力相互作用的理论

应用于宇观尺度/超强引力环境的物理学

→ **广义相对论核心观点（定性）**

质量的存在引起时空的弯曲

在弯曲时空的运动表现为引力

Spacetime tells matter how to move

Matter tells spacetime how to curve

→ **广义相对论的核心公式（定量）**

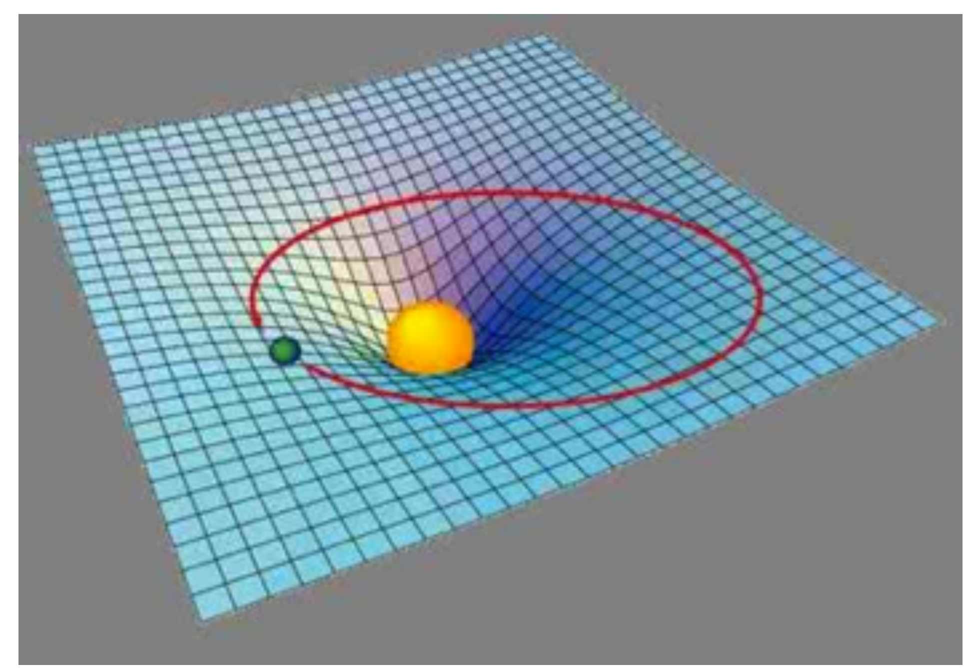
爱因斯坦引力场方程**（装逼必备）**

$$G_{\mu\nu} = R_{\mu\nu} - \frac{1}{2} g_{\mu\nu} R = \frac{8\pi G}{c^4} T_{\mu\nu}$$

→ **这些个带俩下标的东东就是（2阶）张量**

张量的数学表现形式就是矩阵

因为是4维时空，所以是4维张量，即4*4矩阵

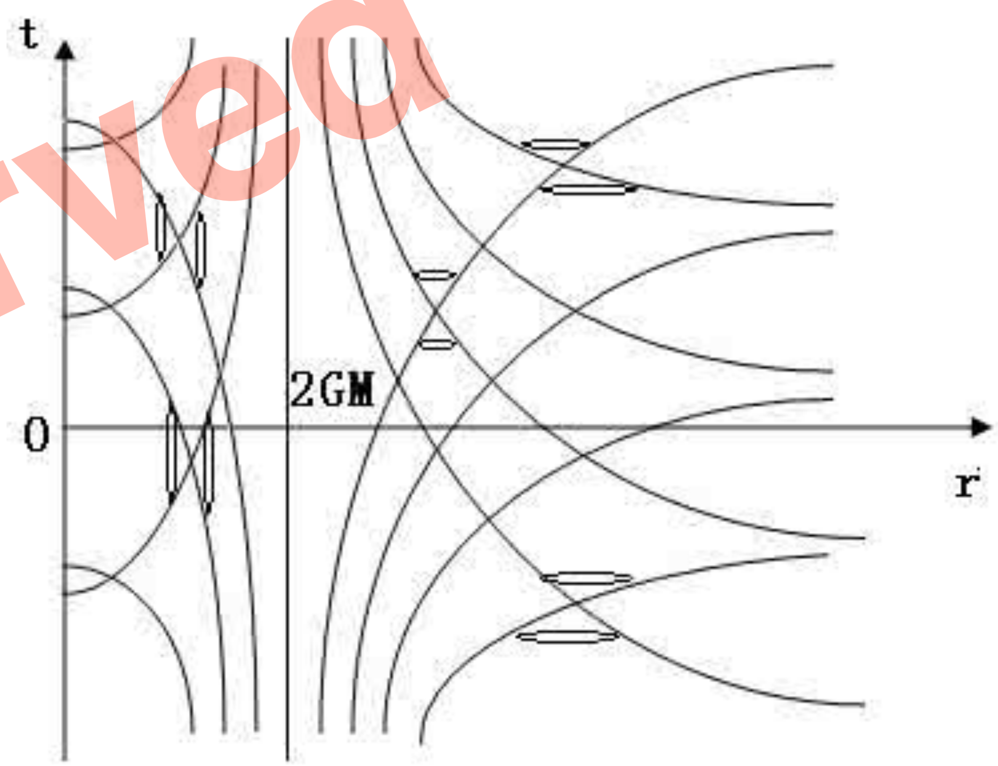


先讲一点科学：黑洞

- 黑洞是时空弯曲超过一定极限导致的特殊**时空结构（场方程的一个解）**
- 什么极限？**时空颠倒**的极限
- **光锥**是对时空结构的形象描述，只有光锥内的时空区域是可达的

从黑洞视界(Horizon)到奇点(Singularity)**不是一段距离，而是一段时间**（不可逆）

普通时空的光锥



黑洞附近的光锥

最小花费问题

n个人中，某些人之间可以互相转账。各人之间转账的手续费率各不相同。给定这些人之间转账时的手续费率，求A最少需要多少钱才能可以转给B转账100元

输入输出格式：

输入第一行两个正整数n,m（n<=2000），分别表示总人数和可以互相转账的人的对数。以下m行每行输入三个正整数x,y,z，表示x和y之间互相转账需要扣除z%的手续费 (z<100)。最后一行输入两个正整数A,B。保证A与B之间可以直接或间接地转账

输出A使得B到账100元最少需要的总费用。精确到小数点后8位

样例输入：

3 3
1 2 1
2 3 2
1 3 3
1 3

样例输出：

103.07153164

图的基本概念

→ **你们已经接触过一些用抽象的点/边描述的数据结构**

链表、树、堆（优先队列），etc

以上数据结构多少对点和边的关系**有一定的限制**

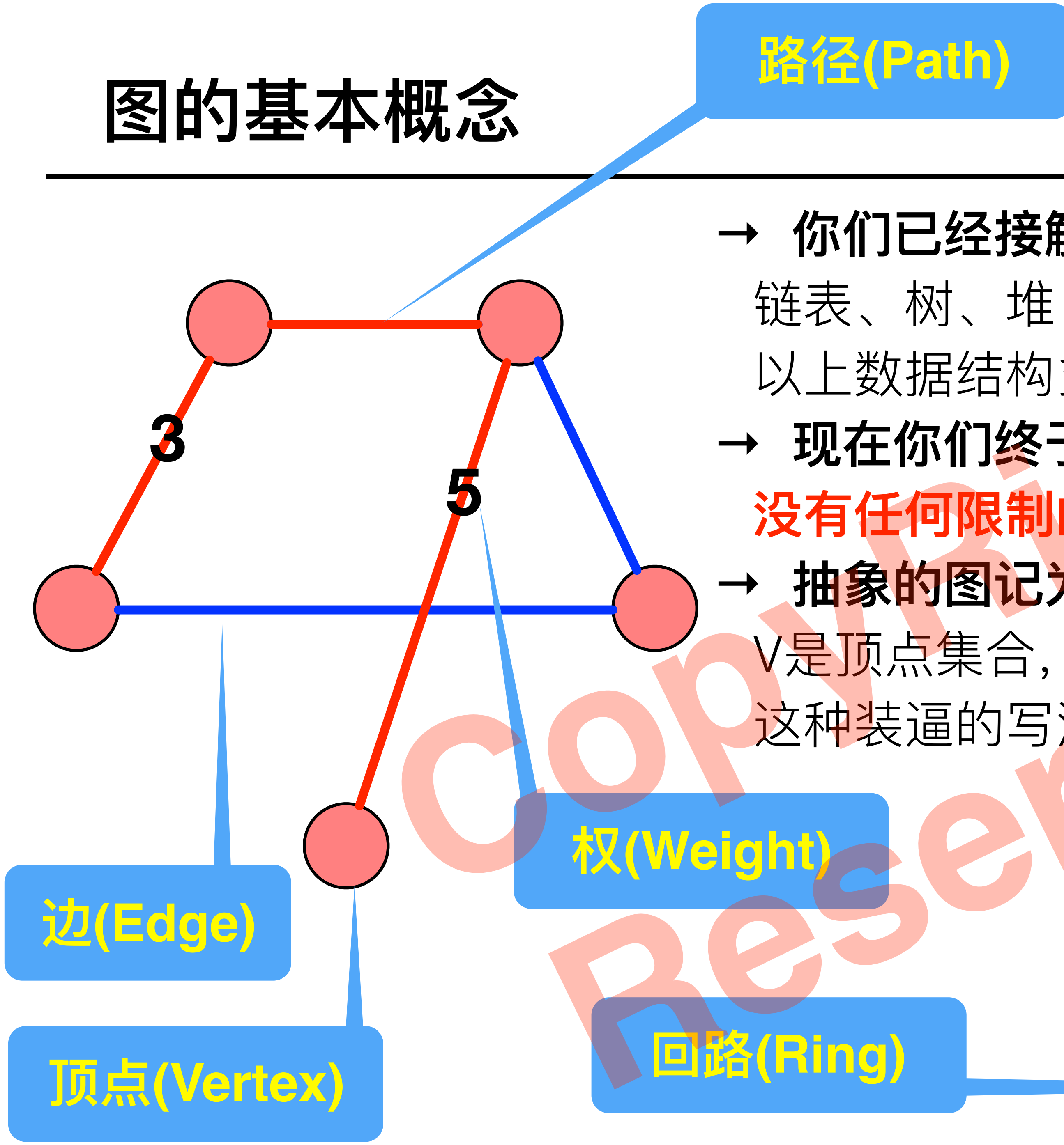
→ **现在你们终于要接触最普遍的情况**

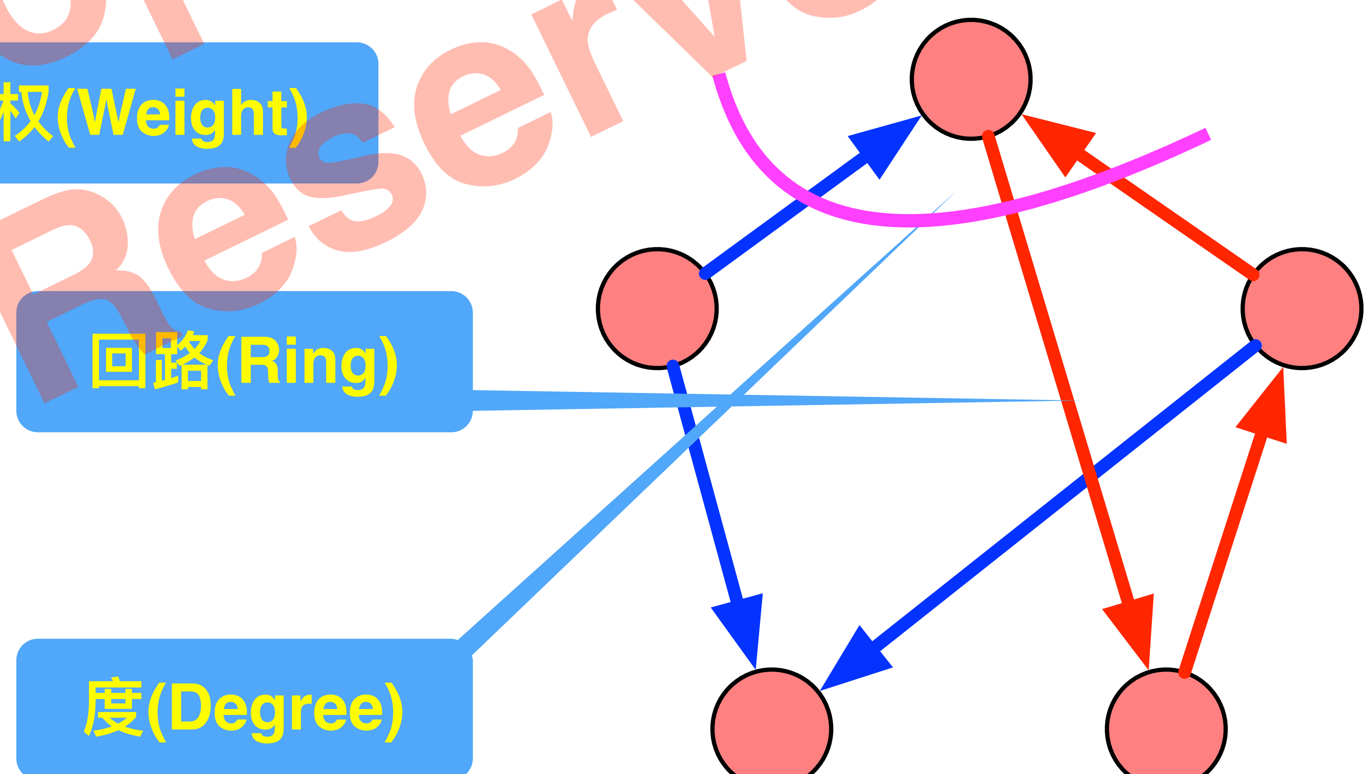
没有任何限制的点与边组成的抽象结构就称为图（Graph）

→ **抽象的图记为G=(V,E)**

V是顶点集合，E是边的集合

这种装逼的写法是为了论述一些问题**书写方便**





图的存储：相邻矩阵

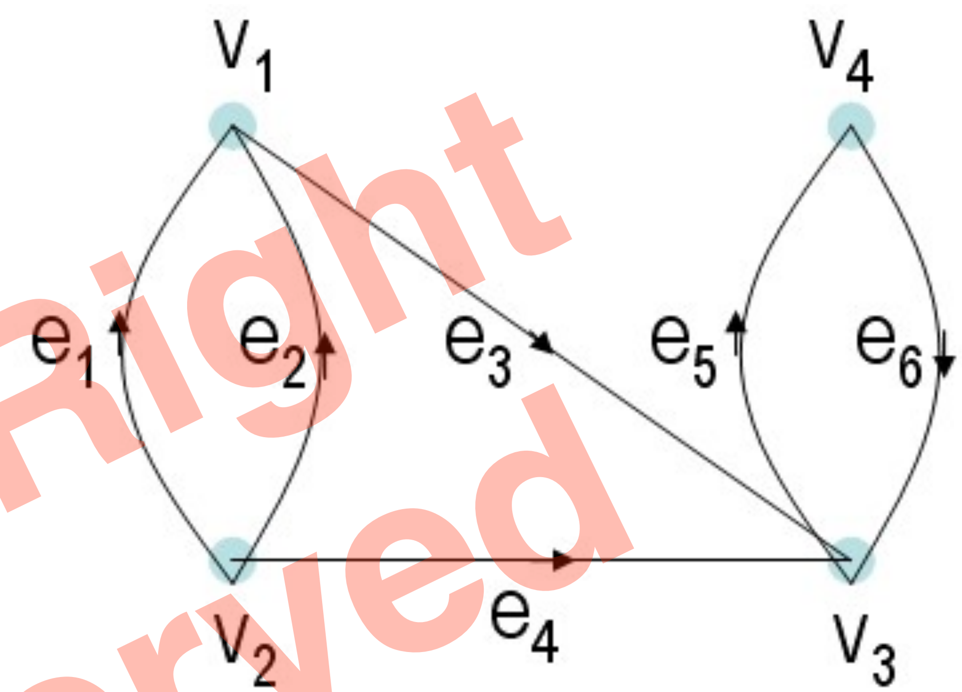
→ **一个n*n矩阵表示一个图**

i,j的值表示i→j边的权值

没有边的顶点对应值为0或∞

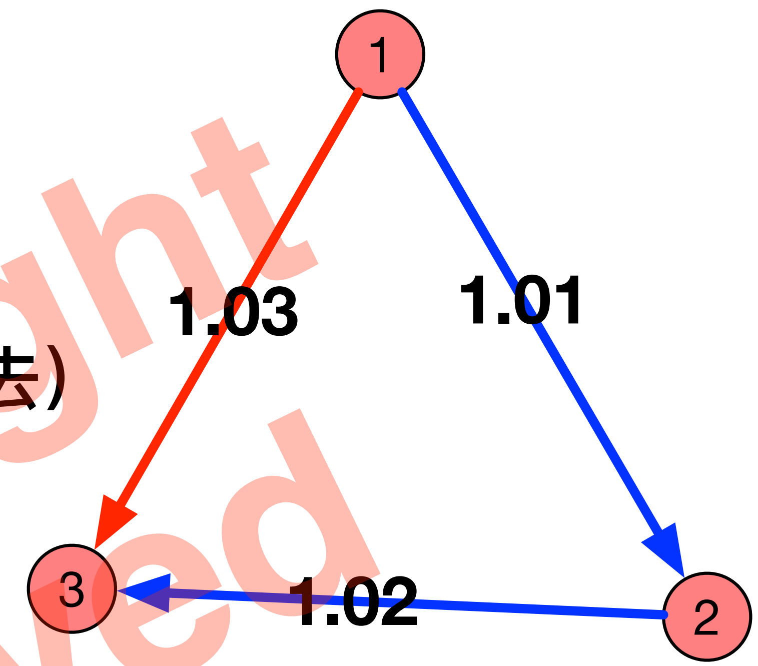
（视情况而定）

→ **如果没有权值可以用0/1表示**



$$M(D)=\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array}\begin{array}{c} \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\ \begin{bmatrix} -1 & -1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \end{array}$$

最小花费问题：建模



→ **图论模型很明显了（人与人之间转来转去）**

顶点：人

边：转账关系

权：手续费率

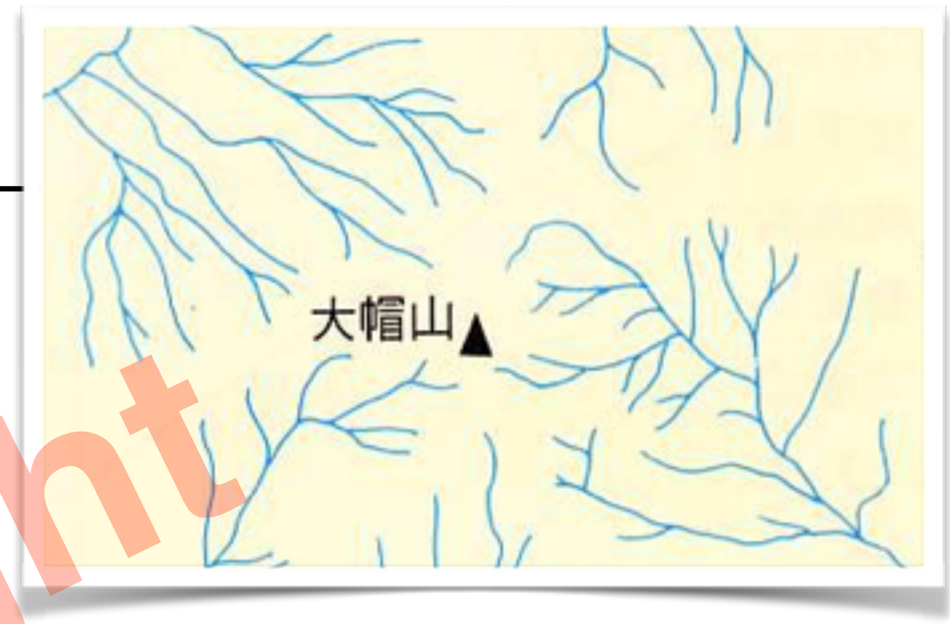
→ **优化目标：求点A到点B的路径，其上权值的乘积最小值**

→ **求顶点之间路径长度最小值，是图论经典问题**

称为**最短路问题**（Shortest Path）

一般路径长度定义为边长之和，这里是乘积，怎么处理？

最短路径：算法思路



→ **假如所有边长均为1?**

用广搜就可以了

→ **进一步，假如所有边长均为正整数**

可将长w的边**截成w条长为1的边**，然后再用广搜

观察原图顶点被搜到的顺序，有什么规律?

→ **再进一步，边长不是整数时**

可将每条边截成**长度灰常灰常小的等长线段**，同理可证

这是一种**从离散到连续**的思维方式

如果你理解了，对以后学习极限/微积分等是有好处的

→ **你可以想象从起点往里灌水**

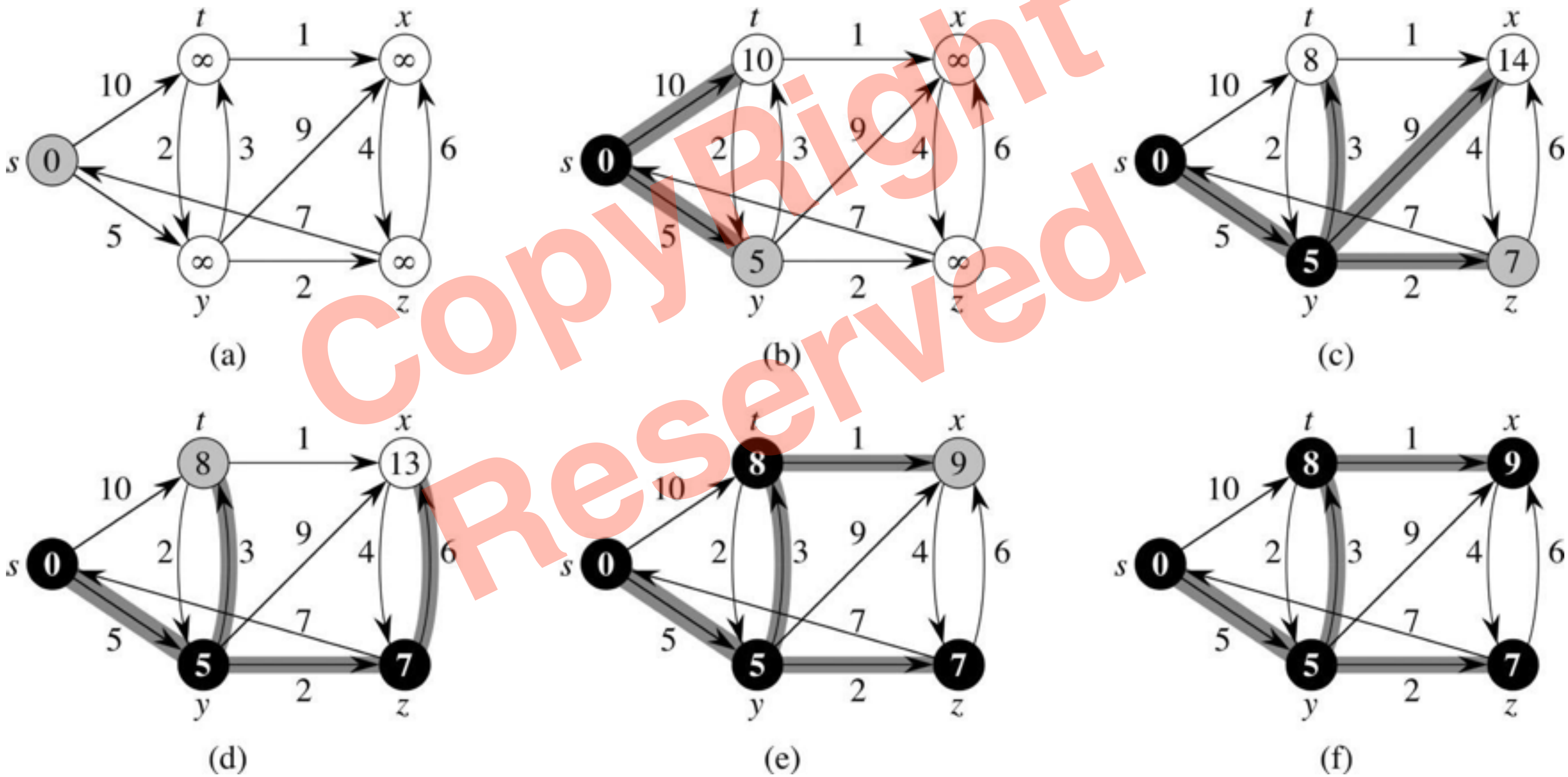
水沿边的方向匀速流动，每淹没一个顶点即为确定其最短路长度

单源最短路：Dijkstra算法

设一个集合X，表示**已经计算出最短路径的顶点**（初始为空）
设一个数组d，用来记录当前（从起点）到某个节点的路径长度
初始时d[a]=0，其他为∞

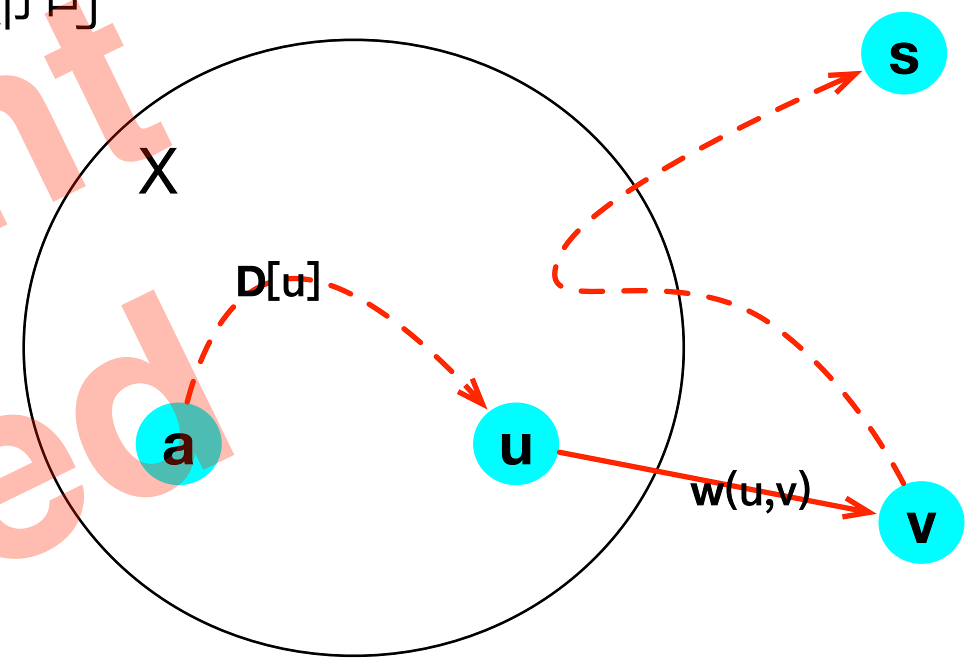
```
for (int i=0;i<n;i++) {
  选择不在X中且d最小的顶点s
  s加入X（认为d[s]就是起点到s的最短路）
  对所有s出发的边 s→t {
    if (t不在X中 && d[s]+w(s,t)<d[t]) {  // w为边长
      d[t]=d[s]+w  // 松弛操作
    }
  }
}
输出d
```

单源最短路：Dijkstra算法演示



Dijkstra算法：证明（装逼版本）

只要证明**每次选中的点s，d[s]就是最短路长**即可



设a→s的**实际**最短路为P

∵a∈X && s∉X中 ∴ ∃(u,v)∈P, s.t. u∈X && v∉X

a沿P到u的距离D[u]就是a→u的最短路长

∵u∈X ∴D[u]=d[u]（X的定义）

u上做过松弛操作（在u加入X时）∴d[v]≤d[u]+w(u,v)

而P是最短路径，所有D[u]+w(u,v)=D[v]

∴**d[v]≤d[u]+w(u,v)=D[u]+w(u,v)=D[v]≤D[s]≤d[s]**

而根据s的选择方式，d[s]≤d[v]

∴需要等号成立，只有d[s]=D[v]=D[s]，

证毕

→ **这些装逼的数学符号**

你到高中/大学就会经常用到

可以节省书写速度，很有用

Dijkstra算法：适用范围

→ 起始点固定，终点任意（单源）

→ 有向图/无向图均可

→ 不能包含负权边（你能看出哪步用到吗？）

→ 本质上是一种贪心算法

→ 你喜欢上面哪种证明?

最小花费问题：解答

```
#define MAXN 2001
#define inf 99999
using namespace std;
int N,M,A,B;
double m[MAXN][MAXN];    // 相邻矩阵

int main() {

    cin>>N>>M;
    for (int i=1;i<=N;i++)
        for (int j=1;j<=N;j++)
            m[i][j]= i==j ? 1 :1/inf;    // 用无穷大表示没有边
    for (int i=0,x,y,t;i<M;i++) {
        cin>>x>>y>>t;
        m[x][y]=m[y][x]=max(m[x][y],1-t/100.0); // 换算费率
    }
    cin>>A>>B;
```

→ **怎么处理边与边之间是乘法?**

1.取对数

2.Dijkstra算法的证明对乘法一样有效
你可以回去自己模仿证一遍

最小花费问题：解答

```
double d[MAXN]; // 最短路径长度（目标值）
bool visited[MAXN]={};     // 访问标志（是否已算出最短路）
for (int i=1;i<=N;i++) d[i]=inf+1; // 初始距离
d[A]=100.0;

for (int j=1;j<=N;j++){
    int s=0;
    double dmin=inf+1;
    for (int i=1;i<=N;i++){
        if (d[i]<dmin && !visited[i]) { // 选择未算出的顶点中dis最小的
            s=i; dmin=d[i];
        }
    }
    visited[s]=true; // 该顶点加入已算出集合
    for (int t=1;t<=N;t++){
        if (d[s]/m[s][t]<d[t] && !visited[t]){
            d[t]=d[s]/m[s][t]; // 松弛操作
        }
    }
}

if (d[B]>=inf){
    cout<<-1<<endl;
}else{
    cout.precision(8);
    cout<<fixed<<d[B]<<endl;
}
return 0;
```

→ **复杂度：O(N^2)**

→ **这地方每次循环去找最小元素**

思考下，有什么优化方法?

拉近距离

在lester和女神jeslie的生活中有N个关键的节点和M个事件，每个事件记为一个三元组（Si，Ti，Wi），表示从节点Si有一个事件可以转移到Ti，效果是使他们间的**距离减少Wi。**这些节点构成了一个网络，其中节点1代表lester，N代表jeslie，其他代表进展的阶段。所有事件可以自由选择是否进行，但**每次只能进行当前节点邻接**的。请你帮他们写一个程序，计算出他们之间可能的最短距离

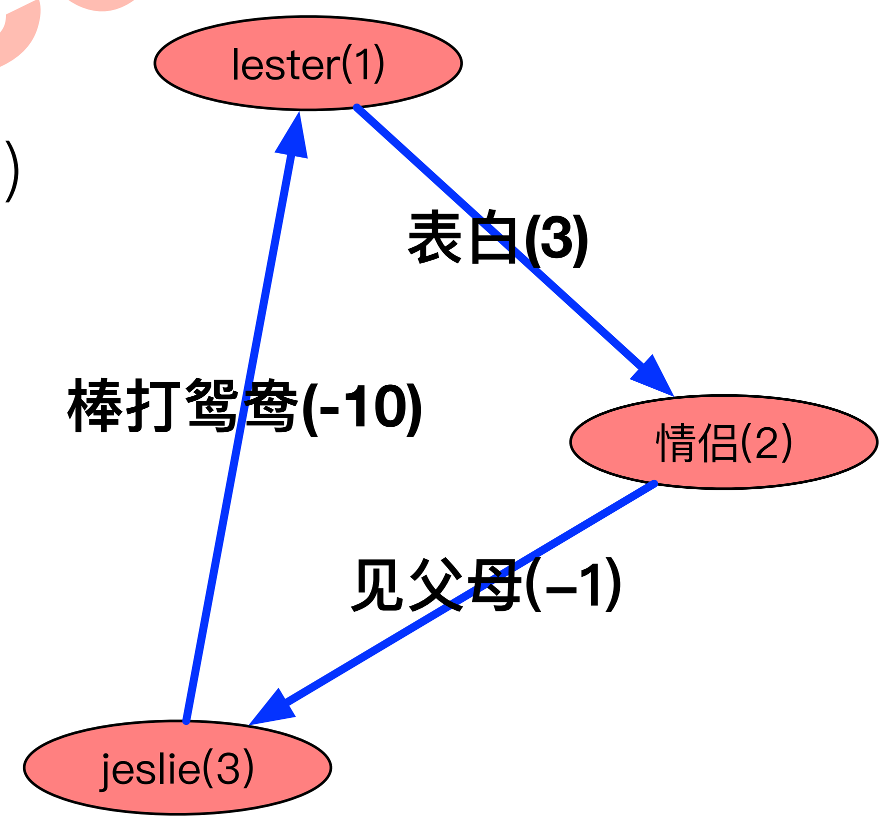
输入输出格式：

输入第1行两个正整数N（<=1000）,M（<=10000）

之后M行每行3个正整数Si，Ti，Wi（绝对值<=100）

输出一个整数表示他们之间可能的最短距离

如可以无限缩小，输出Forever love



样例输入：

3 3
1 2 3
2 3 -1
3 1 -10

样例输出：

-2

拉近距离：建模

- → **图论建模很明显喽**

  以事件为边，权为-Wij，

  目标就是求最短路

- → **存在负权边，所以Dijkstra算法不行orz…**

  发现M比N²小很多，用相邻矩阵很浪费空间（矩阵会很**稀疏**）

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 4 \end{pmatrix}$$

稀疏图的存储：邻接表

- 当边数很稀疏时（远小于O(n2)），可采用如下方式存储图

  为每个顶点创建一个链表

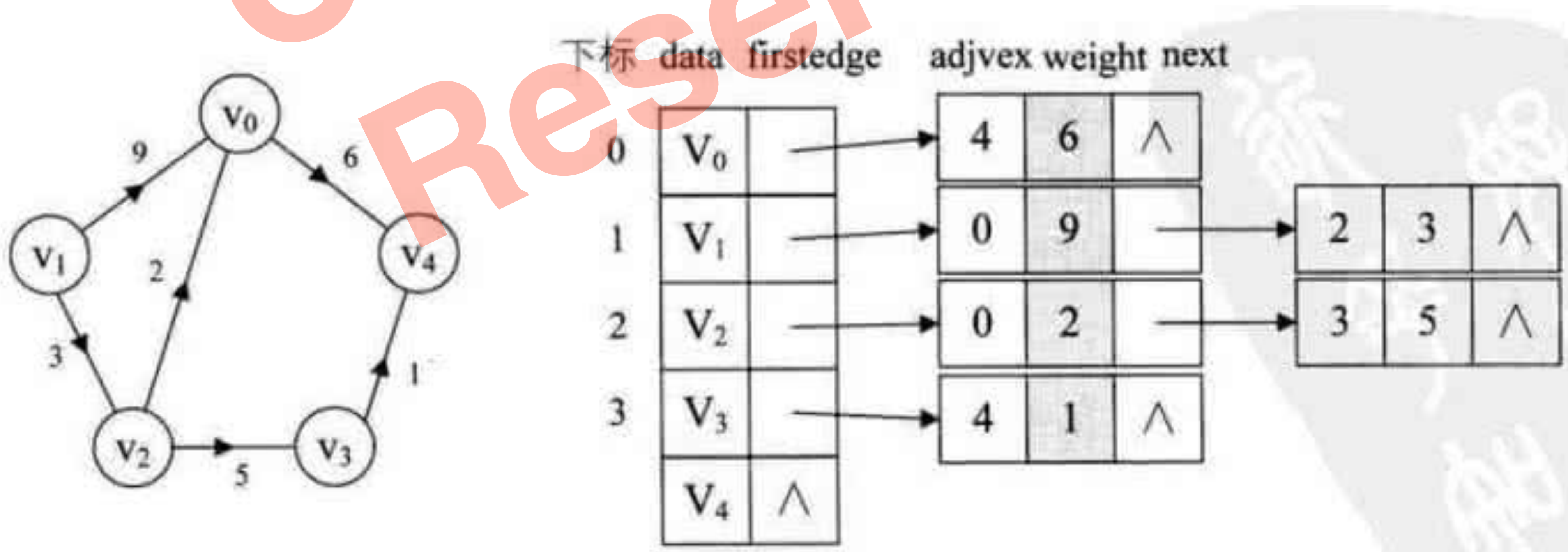
  链表的每一项表示该顶点的一条**出边**

  （链表长度=该顶点的出度）

- 此种存储方式称为图的**邻接表**

  空间性能为O(n+m)，基本没有空间浪费

  支持自环/重边



邻接表：实现（数组方式）

```
const int N=1005,M=10005,INF=0x3f3f3f3f;    // 另一种定义常量的方式

// head和nxt构成邻接表（数组实现版本，nxt其实就是链表指针）
int n,m,d[N],u;
int t[M],w[M]; // 边的信息w为边长，t为每条边的末顶点
int nxt[M],head[N]; // 邻接表，这是一个用数组实现链表的方式，head是头结点，nxt是向后指针
int cnt[N]={},inq[N]; // cnt表示入队次数，inq表示是否在队列中

int main() {
    scanf("%d%d",&n,&m);
    for(int i=1,u;i<=m;i++){
        scanf("%d%d%d",&u,t+i,w+i); // 指针方式数组引用
        nxt[i]=head[u]; head[u]=i; // 从头部插入链表
        w[i]*=-1;    // 取反，因为w表示缩短
    }
```

→ **实现链表有多种不同方式**

你现在看到的是数组方式

好处是**性能相对高一些**，缺点是可读性比较差

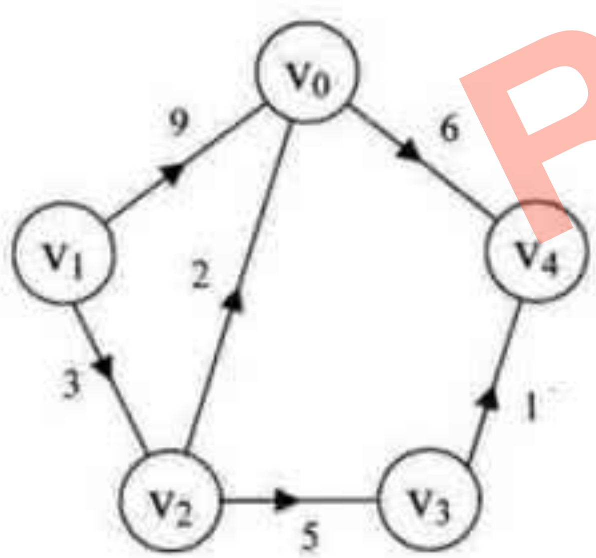
也可以使用指针/vector实现，可读性会更好一些（后面会看到）

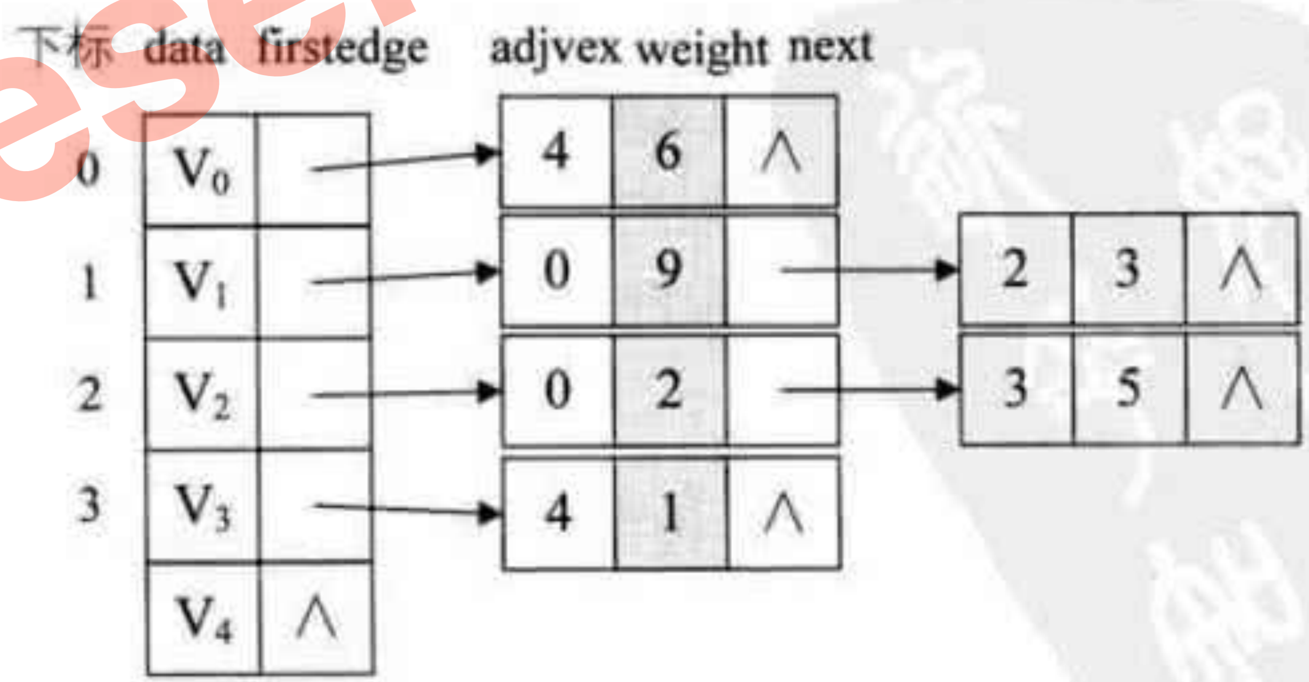
多链的数组方式存储（前向星）

输入顺序

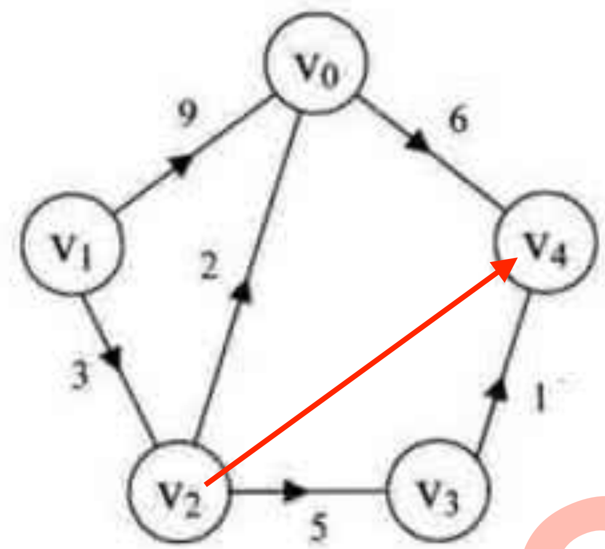
1 0 9
0 4 6
2 0 2
2 3 5
3 4 1
1 2 3

点id	0	1	2	3	4		
head链表首指针	2	6	4	5	0		
边id	1	2	3	4	5	6	7
t:对点	0	4	0	3	4	2	4
w:权	9	6	2	5	1	3	1
nxt:链表后续指针	0	0	0	3	0	1	





增加一条边



点id	0	1	2	3	4		
head链表首指针	2	6	4	5	0		
边id	1	2	3	4	5	6	7
t:对点	0	4	0	3	4	2	4
w:权	9	6	2	5	1	3	1
nxt:链表后续指针	0	0	0	3	0	1	

点id	0	1	2	3	4		
head链表首指针	2	6	4	5	0		
边id	1	2	3	4	5	6	7
t:对点	0	4	0	3	4	2	4
w:权	9	6	2	5	1	3	1
nxt:链表后续指针	0	0	0	3	0	1	4

nxt[7]=head[2]
head[2]=7

负权单源最短路：SPFA算法

设一个数组d，记录当前（从点1）到某个节点的路径长度
初始时d[1]=0，其他为∞
设置一个队列q，存放待处理顶点（初始只有一个元素1）

```
while (q非空) {
  出队顶点s
  对所有s出发的边（s,t）{
    if (d[s]+w(s,t)<d[t]) {
      d[t]=d[s]+w  // 松弛操作
      if (t不在q中）t入队
    }
  }
}
输出d
```

→ **似乎比dijkstra算法还简单**
思想就是**能松弛就松弛**
（本质上还是贪心）
→ **该算法称为SPFA算法**
(Shortest Path Faster Algorithm)

拉近距离：解答

```
for(int i=2;i<=n;i++)  d[i]=INF; // 初始化距离
queue<int> q;
q.push(1); cnt[1]=1; inq[1]=1; // 起点入队
while (!q.empty()) {
    u=q.front(),q.pop(),inq[u]=0;    // 出队
    if(cnt[u]>=n){  // 重复入队超过n次，说明有负权回路
        printf("Forever love\n"); return 0;
    }
    for (int e=head[u];e;e=nxt[e]){ // 遍历e出发的所有边
        if(d[t[e]]>d[u]+w[e]) {
            d[t[e]]=d[u]+w[e]; // 松弛
            if(!inq[t[e]]) {    // 如果不在队列中，则入队
                q.push(t[e]); cnt[t[e]]+=1; inq[t[e]]=1;
            }
        }
    }
}
printf("%d\n",d[n]);
```

SPFA算法：证明

→ 哪条边都不能再松弛了，不就是最小了呗?

如果你这么想，那么你犯了一个概念错误

某种属性对于任何**局部**调整下都不会变小，该状态称为**极小**

注意**极小未必是最小**（反之成立）

→ 正确性证明

边(u,v)满足d(v)-d(u)≤w(u,v)，称该边为**饱和的**（不可松弛）

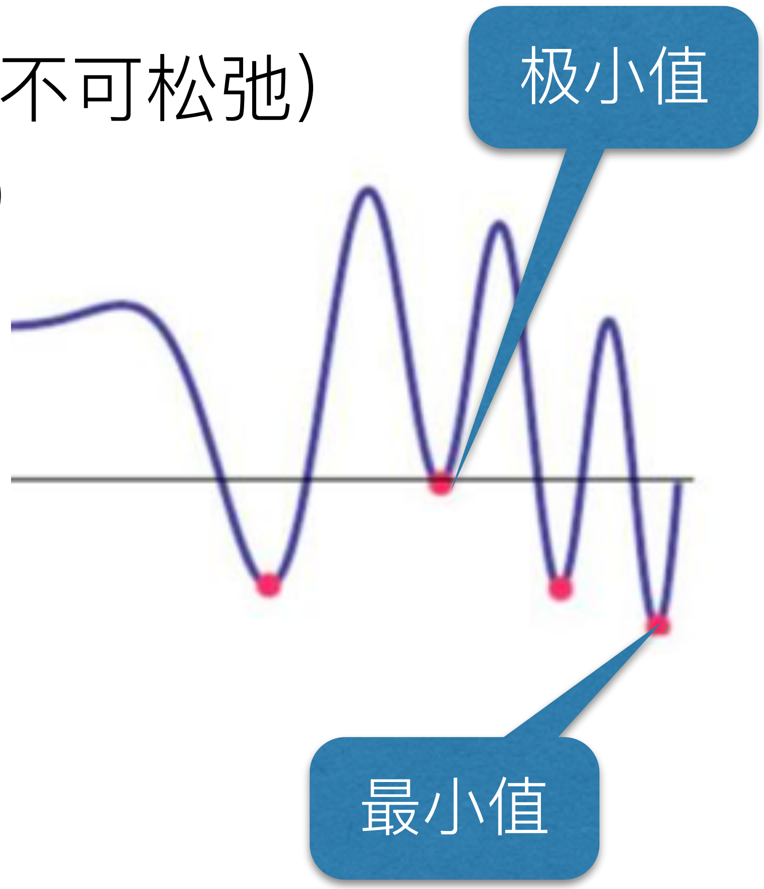
如路径P:x→y上所有边都饱和，则d(y)-d(x)≤w(P)

设s到i**实际**最短路径为Ps，显然d(i)≥w(Ps)

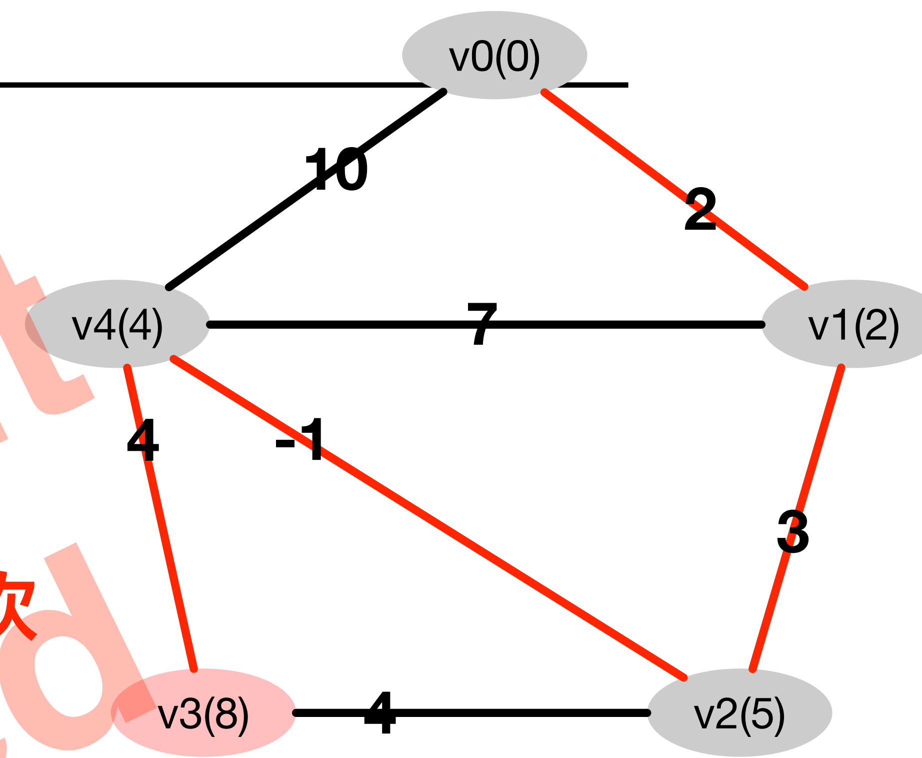
而最短路径上的边一定都是饱和的

于是d(i)-d(s)=d(i)≤w(Ps)

∴d(i)=w(Ps) 证毕



SPFA算法：复杂度（选学）



→ **每个顶点可能入队几次?**

所有入队的顶点按顺序分为多个批次

v0（起点）为第1批次

第i批次全部出队后，队列中剩余的为第i+1批次

（i类似广搜的步长，每批次中没有重复顶点）

→ **所有顶点最短路组成的边集，一定是一颗生成树T**

生成树（Spanning Tree）下周就会讲到，现在你知道是树就行了

→ **T中深度为k的顶点最多经过k批次后就找到最短路（归纳法）**

k=0，就是起点，Trivial

k>0，其父节点在在前k-1批次中找到最短路

父节点再松弛一次（下一批次），则当前节点也找到最短路。证毕

→ **最多有n个批次**

SPFA算法：复杂度（选学）

→ **每批次中每个顶点最多出队一次**

因此**每批次每条边最多松弛一次，**复杂度不超过O(m)

→ **总复杂度不超过O(nm)**

→ **通常情况达不到此上限，一般在2*m~3*m左右**

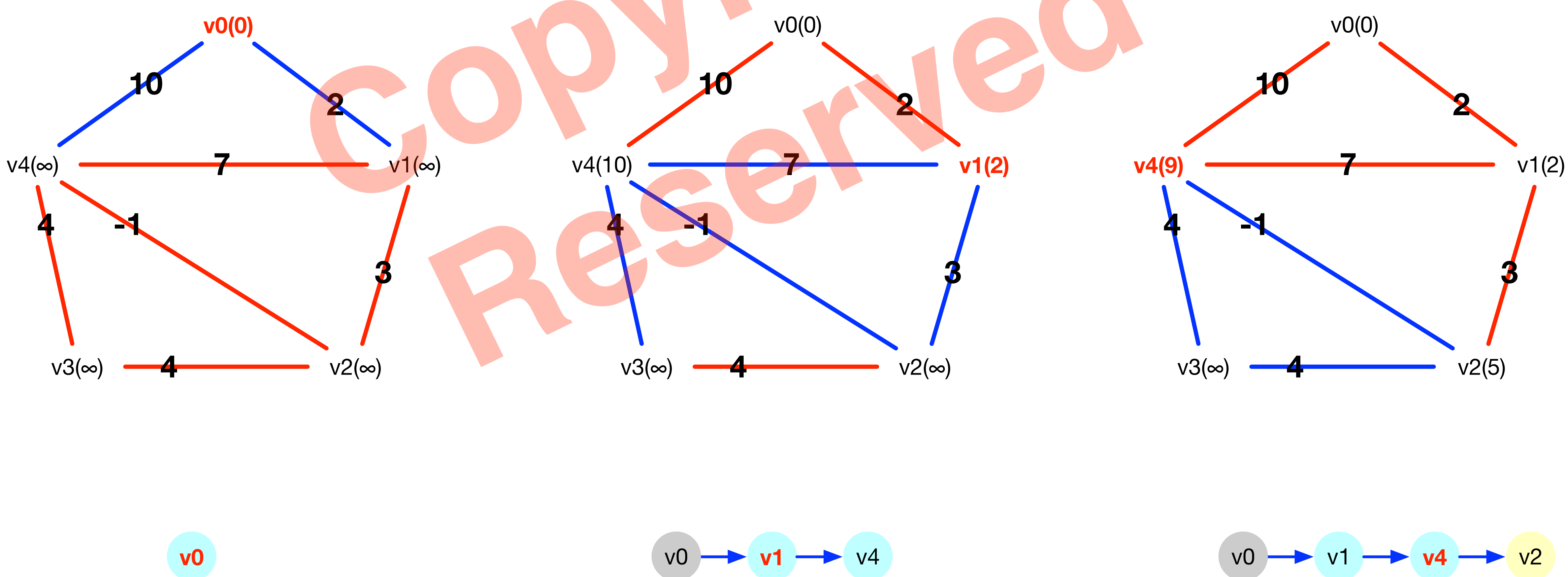
感兴趣的自己去网上查查SPFA复杂度的讨论

SPFA算法：演示（选学）

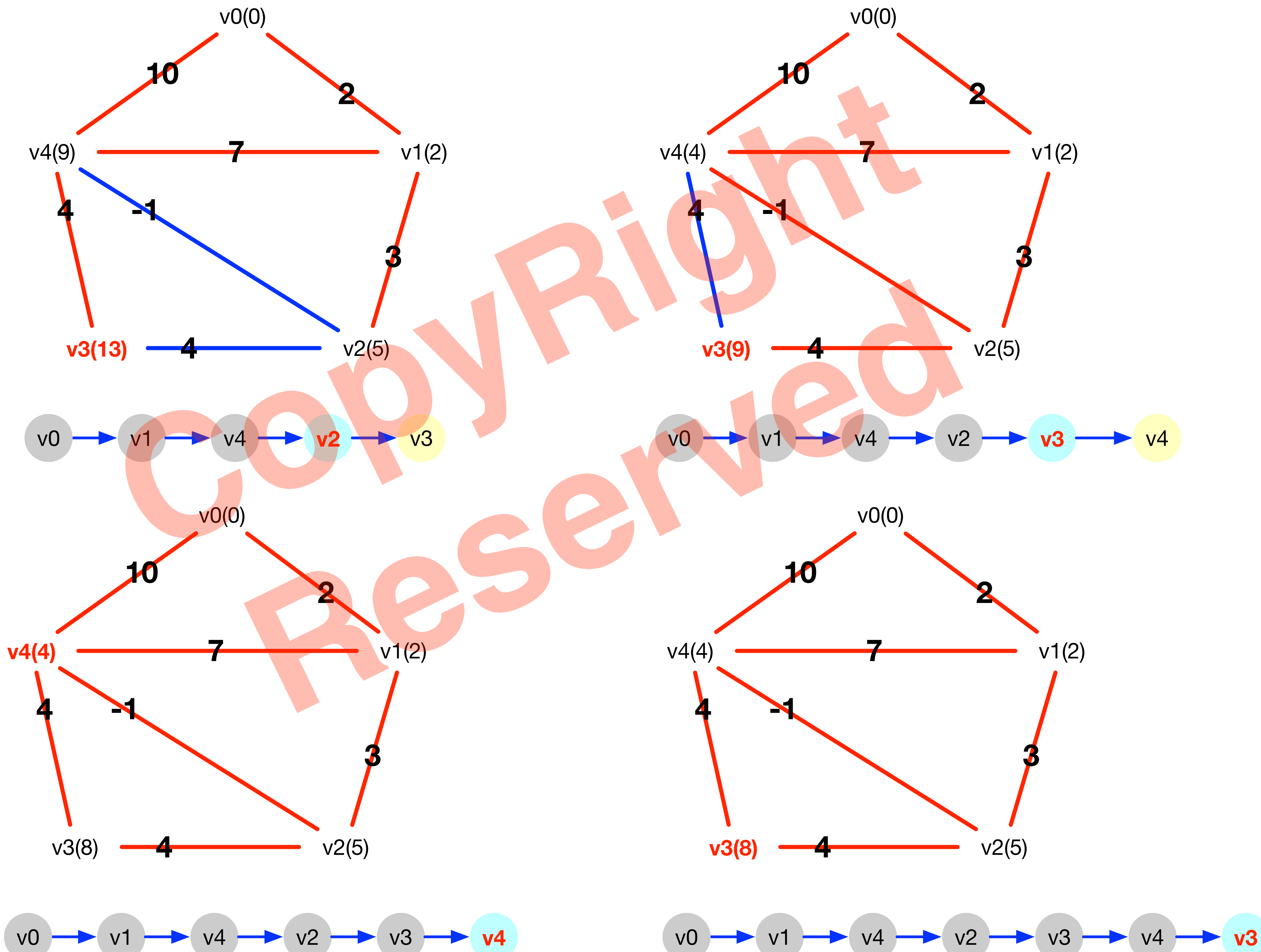
→ 括号内为d的值，红色字体为当前处理节点

→ 红色边为饱和边，蓝色边为可松弛

→ 蓝圈为当前批次，黄圈为下一批次，灰圈为已出队



SPFA算法：演示（选学）


v0(0)
10
2
v4(9)
7
v1(2)
4
-1
3
v3(13)
4
v2(5)
v0
v1
v4
v2
v3
v0(0)
10
2
v4(4)
7
v1(2)
4
-1
3
v3(9)
4
v2(5)
v0
v1
v4
v2
v3
v4
v0(0)
10
2
v4(4)
7
v1(2)
4
-1
3
v3(8)
4
v2(5)
v0
v1
v4
v2
v3
v4
v0(0)
10
2
v4(4)
7
v1(2)
4
-1
3
v3(8)
4
v2(5)
v0
v1
v4
v2
v3
v4
v3

SPFA算法：适用范围

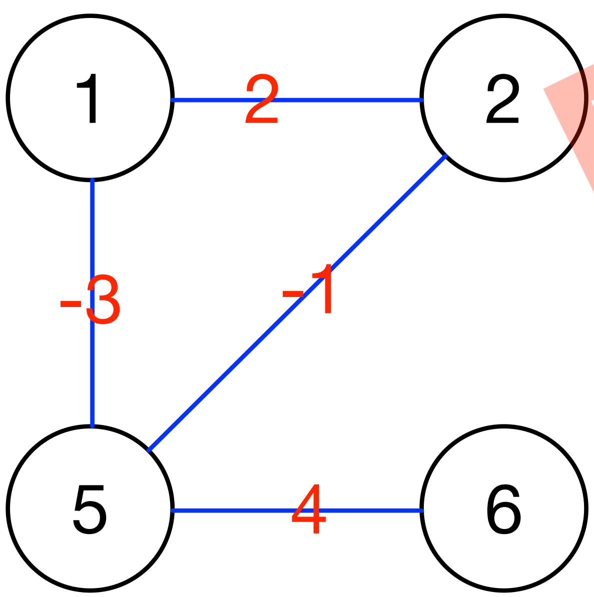
- → **可以有负权边**
- → **有向图/无向图均适用**
- → **不能有负权回路**

如果有的话，也就不存在最短路（因为可以无限短）

如有节点入队超过n次（队列仍未空），说明一定存在负权回路

（此种情况下SPFA算法达到最坏复杂度O(nm)）

也可用于**负权回路存在性判定**



→ **在相对论中，时空的负权回路称为类时闭曲线**

所谓**时间机器**就是指寻找类时闭曲线

作业

1.dijkstra算法无法处理带负权边图的最短路径，请举一个反例

（设计一个图，使得Dijkstra算法得出的非最短路径。

Note：不允许有负权回路，否则就没有最短路径了）

2.（选做）松弛顺序

SPFA中，边的松弛顺序并没有严格要求

之前我们用了队列（先进先出）

你也可以用栈（先进后出）

也可以用优先队列（你会发现转化为Dijkstra算法）

你觉得如何选择松弛的顺序更好?

3.请柬分发

喜剧演员lester想要宣传剧院，许多学生被雇来分发请柬。每个学生被指定一个确切的公共汽车站，他或她将留在那里一整天，邀请人们参与。公交系统所有的线路都是**单向的**，连接两个站点。公共汽车离开起始点，到达目的地之后又**空车返回**起始点。学生早上从总部出发，乘公交车到预定站点。每个站点都被安排了一名学生。一天结束时，所有学生都回到总部。求学生所需的公交费用总和最小值

输入输出格式：

输入第1行有两个整数n、m(1<=n,m<=1000000)，n是站点的个数，m是线路的个数

然后有m行，每行描述一个线路，包括3个整数，起始点，目的地和价格

总部在第1个站点，价钱小于1000000000

输出一行，表示最小费用

样例输入：

4 6
1 2 10
2 1 60
1 3 20
3 4 10
2 4 5
4 1 50

样例输出：

210

4.（选做）最短路计数

一个N个顶点M条边的无向无权图，求从顶点1到其他各点最短路各有几条

输入输出格式：

输入第一行2个正整数N（<=1000000），M（<=2000000），为顶点与边数

接下来M行，每行两个正整数x, y，表示有一条顶点x连向顶点y的边，请注意可能有自环与重边

输出一行N个非负整数，第i个为从顶点1到顶点i有多少条不同的最短路，只需输出mod 100003后的结果。无法到达的点输出0

样例输入：

```
5 7
1 2
1 3
2 4
3 4
2 3
4 5
4 5
```

样例输出：

```
1 1 1 2 4
```