图论 ---- 最短路径

一.前言

说起图论算法，最经典的莫过于最短路径了。它永远是图论难以绕开的经典问题，也常常是学习图论遇到的容易卡住的难点。
最短路径，那首先研究的对象就是两点间的路径，它要有起点和终点。
图中两个连通结点之间至少存在一条路径，**最短路径就是其中权值和最小的路径。**

算法中对权值定义可以很广泛，它可以是点权，也可以是边权，也可能多关键字的权重...总之不论权值如何定义，重要的只是权值和能够被表示出来且能够被比较就可以了。

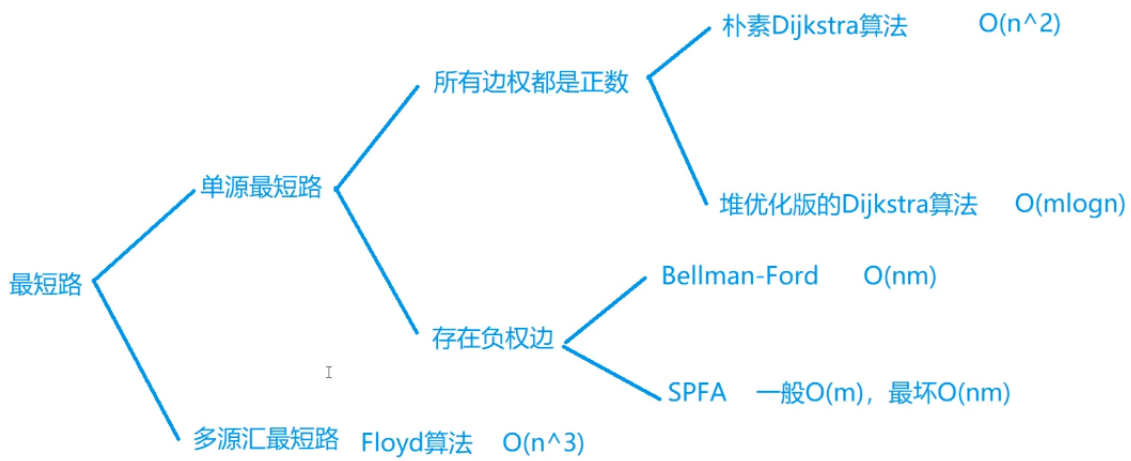
探讨的最短路径分为两种：单源最短路径和 多源最短路径

多源最短路径 ---------任意两点间的最短路径
单源最短路径 ---------指定两点间的最短路径

我们本次会介绍4种最常用也是最常考的算法，如下

1. floyed算法
2. dijkstra算法 【暴力 + 堆优化】
3. Bellman-Ford算法[直接学SPFA算了]
4. SPFA算法

常用的规律是： **正权图可以使用floyed算法 + dijkstra算法 , 负权图请使用Bellman-Ford算法 + SPFA算法+floyed算法**



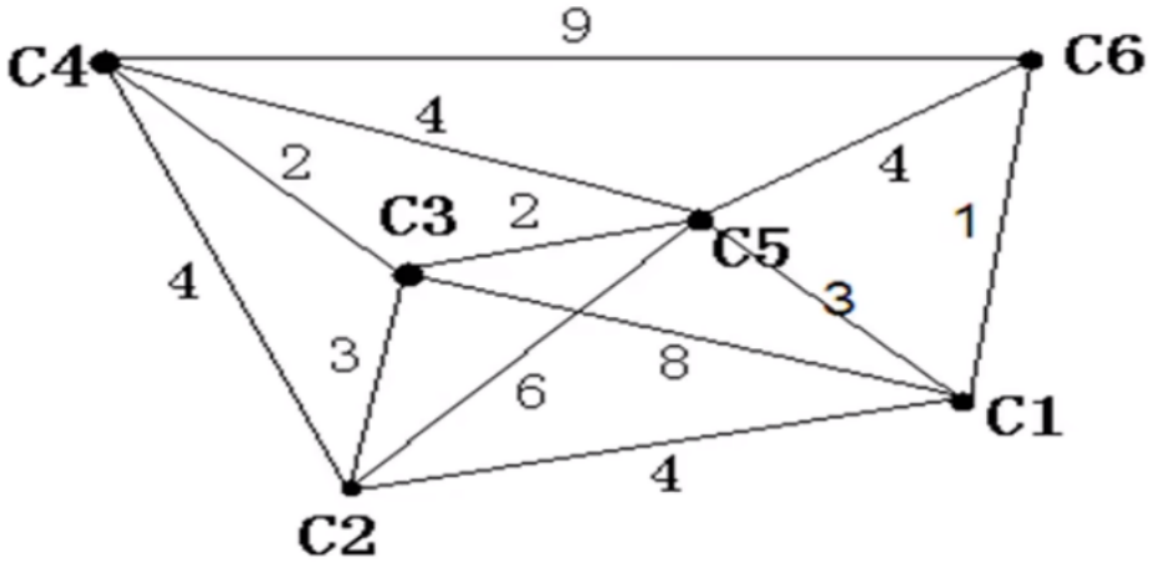
本次我们先讨论权值为正数的正权图。

目前常见的有两种经典算法：**dijkstra算法**和**floyed算法**

二. Floyed算法

Floyed 算法（弗洛伊德算法）是最简单的最短路径算法，可以计算图中**任意**两点间的最短路径。

算法原理



1. 从任意一条单边路径开始。所有两点之间的距离是边的权，如果两点之间没有边相连，则权为**无穷大。**
2. **对于每一对顶点 u 和 v，看看是否存在一个过渡点 w ，使得从 u 到 w 再到 v 比已知的路径更短。**

   如果是更新起点到终点的距离， 否则直接跳过。

```
if( e[u][v]>e[u][过渡点]+e[过渡点][v] )  e[u][v] = e[u][过渡点] + e[过渡点][v];
```

3. 最终如果两点距离为无穷大，则证明此两点不连通

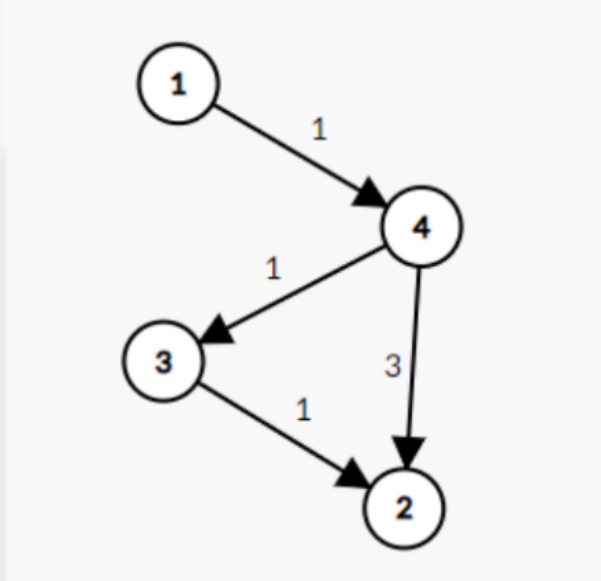
Floyed算法核心

```
// 三重for循环
for(int k=1;k<=n;k++){// 先遍历所有的过渡点
    for(int i=l;i<=n;i++){ // 再遍历所有的起点
        for(int j=1;j<=n;j++){ // 最后遍历所有的终点
            if(e[i][j]>e[i][k]+e[k][j]){
                e[i][j]=e[i][k]+e[k][j];
            }
        }
    }
}
```

小细节注意

上述的三重for循环，需要先去遍历所有的过渡点，再遍历起点，最后遍历终点
很多人会顺手写成先起点，后终点，最后过渡点，注意此方法会WA，证明如下

比如我们来看一个例子：



当 $i=1,j=2,k=3$ 时，此时 $dis_{1,3}=+\infty$，无法更新到 $dis_{1,2}$。

到 $k=4$ 的时候，将 $dis_{1,2}$ 更新为 $dis_{1,4}+dis_{4,3}=4$。

然后在 $i=1,j=3,k=4$ 时，更新 $dis_{1,3}=2$，但无法在将 $dis_{1,2}$ 更新为 $dis_{1,3}+dis_{3,2}=3$。所以在最终结果中，$dis_{1,2}=4$。

优缺点分析

时间复杂度：O(n^3)
空间复杂度：O(n^2))

Floyd算法适用于APSP(All Pairs Shortest Paths,多源最短路径）,是一种动态规划算法，稠密图效果最佳，边权可正可负。
此算法简单有效，由于三重循环结构紧凑，对稠密图，效率要高于执行M次Djkstra算法，也要高于执行M次SPFA算法。
优点：容易理解，可以算出任意两个节点之间的最短距离，代码编写简单。
缺点：时间复杂度比较高，不适合计算大量数据。

综上所述，会用即可，个别题目可以用来混分，对于恐怖的二维数组，再加上更恐怖的三重for循环，实际应用意义不大，但是，但是，但是，关键的来了，如果你发现点的个数非常小，果断Floyed绝对没问题

完整代码展示

```
#include<bits/stdc++.h>
using namespace std;
int n , m; // n点  m边
long long a[4505][4505] , p , q , r;
int main(){

    cin >> n >> m;
    // 预处理
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++){
            if(i!=j) a[i][j]=INT_MAX;
        }
    }
    for(int i=1;i<=m;i++){
        cin >> p >> q >> r;
        a[p][q] = a[q][p] = min( a[p][q] , r ); // 重边
    }
    // Floyed算法
    for(int k=1;k<=n;k++){
        for(int i=1;i<=n;i++){
            for(int j=1;j<=n;j++){
                a[i][j] = min( a[i][j] , a[i][k]+a[k][j] );
            }
        }
    }
    // 输出
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++){
            cout << a[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

三. dijkstra算法

PS : 需要前置知识点---------> 链式前向星 + 优先队列[堆]

算法原理 + 理解方案

对于一个算法，首先要理解它的**运行流程。**
对于一个Dijkstra算法而言，前提是它的前提条件和环境：

- 一个连通图，若干节点，节点可能有数值，但是路径一定有权值。并且路径**不能为负**，否则Dijkstra就不适用。
- **单源最短路径**

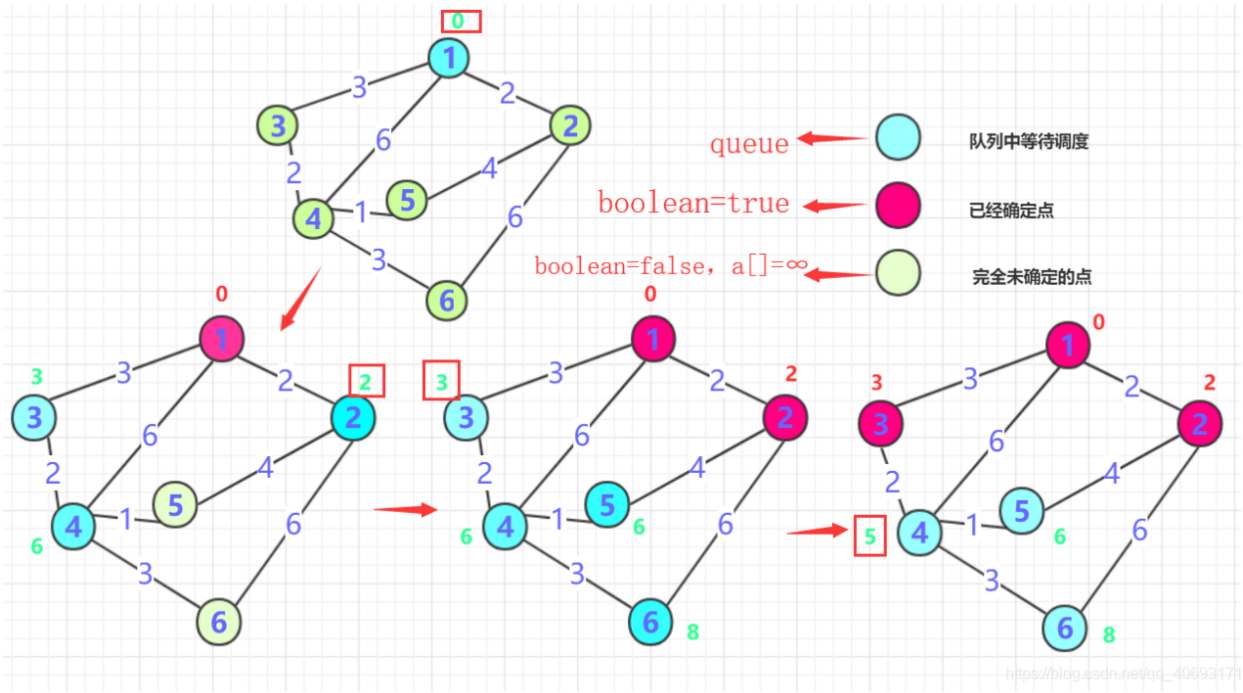
Dijkstra的核心思想是贪心算法的思想，它每次会选取一个离源点最近且还未被选取的点进行选取，总共进行n次选取最终可以选取到dis[n]的最短距离。

我们把点分成两类,一类是已经确定最短路径的点,称为"黄点",另一类是未确定最短路径的点,称为"蓝点"

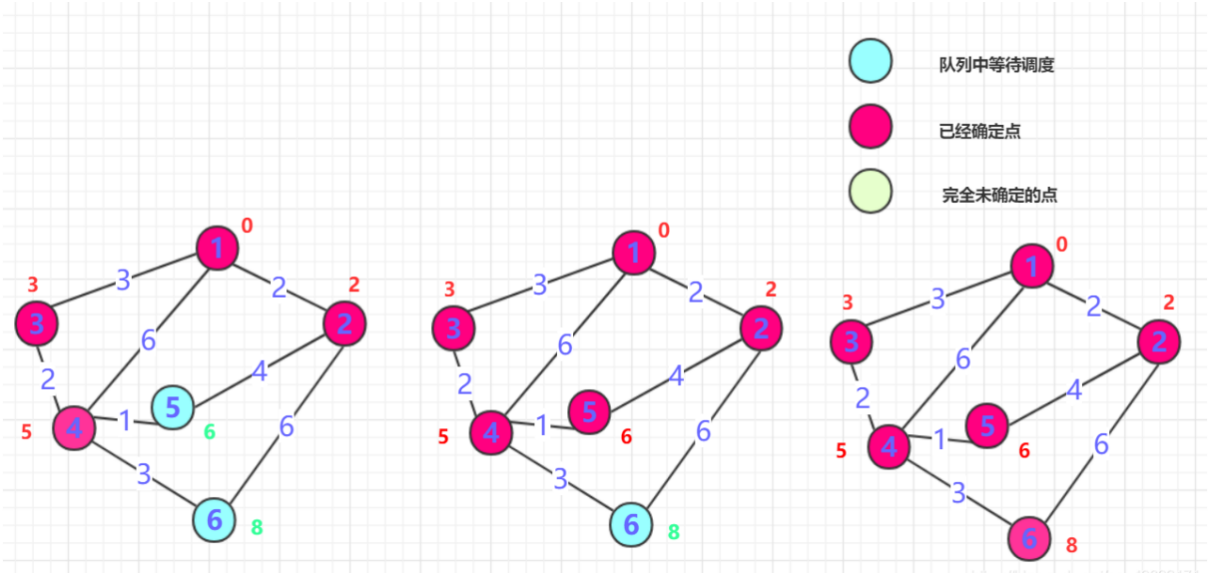
dijkstra的流程如下:

1. **初始化所有节点的dis值为无穷大，然后将dis[起点] = 0，.**
2. **找一个当前dis值最小的蓝点x , 把节点x变成红点.**
3. **遍历x的所有出边(x--y值为z) , 若dis[y] > dis[x] + z则令dis[y] = dis[x] + z**
4. **重复2,3两步,直到所有点都成为红点..**

· 时间复杂度为O(n^2)



- 重复二的操作，直到所有点都确定。

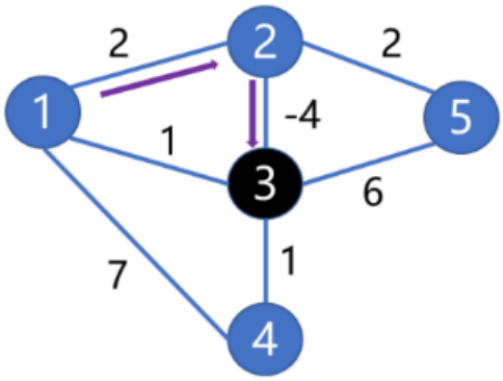


首先采用的是贪心的策略，各位都能看出来，找到当前权值最小的且为0的点，再进一步扩充它的最近的点，更新每个点最短距离，不断地寻找中间结点w使得u → w → v 的距离小于u → v的距离，即用中间值更新结果再找最小的点【松弛操作】，再按照上述策略继续重复，直到所有的点都被更新过即可。

小细节注意

必须路径**不能为负**，即不能出现负边权。

Dijkstra 算法不能处理存在负边权的情况；



因为Dijkstra算法是每次选取到这个合法的点后会将此点的最短距离进行确定下来，它会默认这个点已经是最优解了，而可以进行这个默认的前提就必须是边权为正值，因为如果存在边权为负值则可能导致之后再次使用另一个点 y 去优化的过程将已经确定的点 x 的dis[x]的值可能再次减小，这就和这个算法的原理相违背了，不能确保每一次选取的点为最优点。

Dijkstra算法两种版本

1. 传统版本：

1. 找到未被选择的点当中离起点最近的点 x，找dis最小的那个点
2. 确定这个点 x 的值即dis[x] ，定死dis[x]值即最终起点到x的最短路径的值，并标记
3. 利用点 x 去循环遍历 x 和其他未被访问的点的距离。

重复这3个过程 n 次即可找到 n 个点到源点的最短路了。
因为我们是利用点与点之间的关系进行遍历的一共进行 n 次，每次需要遍历 n 个点 所以时间复杂度： **O(n*n)**
【so 这里隐藏着TLE的分险，所以后面需要优化】

```
#include<bits/stdc++.h>
using namespace std;
int dis[100001]; // dis[i]代表出发点到达i点的最短路径
int visit[100001]; // 是否以当前点作为中心点进行计算过
int first[100001];// 储存当前点i为起点的最后一条边
int n,m,s; // n个点  m个边  出发点s
//链式前向星
struct node{
    int to; // 当前边的终点【子节点】
```

```
    int w;  // 当前边的权值
    int next; // 以当前边为起点【父节点】的上一条边
}edge[500001];

int cnt; // 处理的第几条边【边的编号】
void add(int u , int v , int w){ // 起点u  终点v  权值w
    edge[++cnt].to = v;
    edge[cnt].w = w;
    edge[cnt].next = first[u];
    first[u] = cnt;
}
int p,q,r; // 临时边 ---起点终点权值
int main(){

    cin >> n >> m >> s;
    while(m--){
        cin >> p >> q >> r;
        add(p , q , r);
    }
    // dijkstra算法
    for(int i=1;i<=n;i++) dis[i]=INT_MAX;
    dis[s] = 0;
    for(int i=1;i<=n;i++){
        int mmin = INT_MAX;
        for(int j=1;j<=n;j++){
            if( visit[j]==0 and dis[j] < mmin ){
                mmin = dis[j];
                u = j;
            }
        }
        visit[u] = 1;
        // 以下dis最小值的那个点是u
        for(int j=first[u] ; j!=0 ; j=edge[j].next){
            v = edge[j].to;
            if( dis[v] > dis[u] + edge[j].w ){
                dis[v] = dis[u] + edge[j].w;
            }
        }
    }
    for(int i=1;i<=n;i++) cout << dis[i] << " ";

    return 0;
}
```

2. 堆优化迪杰斯特拉算法

朴素Dijkstra的优化主要就出在找最小值上，外层O(N)的循环是无法优化的，这一步由于每次都要遍历一遍所有点，复杂度为O(n)

不过可以用堆（STL：优先队列）进行优化，复杂度为O(logn)

优化查找过程，首先初始化堆只有源点，初始化dist[] = INF，除了dist[x]=0，然后每次从堆中取出最小距离的中间结点j并出堆，将所有能更新的 dist[j] + map[j] [k] < dist[k] 的 dist[k] 更新为 dist[j] + map[j] [k] ，直到整个堆清空为止。

PS :**由于朴素算法是用数组编号来映射结点的，而堆排序会打乱原映射顺序，所以我们考虑用C++自带的pair<typeA , typeB>来存储结点，由于pair默认按照typeA排序，所以typeA应该是距离，typeB是结点编号。**

```
#include<bits/stdc++.h>
using namespace std;
int n,m,s;// n点  m边  起点s
int u,v,w,cnt;// cnt边号
struct node {
    int to,w,next;
} edge[200005];
int first[100005];
int visit[100005]; // 是否被访问
int dis[100005];
// 添加边
void add(int u,int v,int w) {
    edge[++cnt].to = v;
    edge[cnt].w = w;
    edge[cnt].next = first[u];
    first[u] = cnt;
}
// 小根堆 --- 前面int代表距离，后面int代表当前点
priority_queue< pair<int,int> , vector<pair<int,int> > , greater<pair<int,int> > >
que;
int main() {

    cin >> n >> m >> s;
    for(int i=1; i<=n; i++) dis[i]=INT_MAX;
    while(m--) {
        cin >> u >> v >> w;
        add(u,v,w);
    }
    dis[s] = 0;
    que.push(  make_pair(0,s) );
    while( !que.empty() ) {
        u = que.top().second;
        que.pop();
        if(visit[u]==1) continue;
        visit[u] = 1;
        for(int i=first[u] ; i!=0 ; i=edge[i].next) {
            v = edge[i].to;
```

```
            if( dis[v] > dis[u] + edge[i].w ) {
                dis[v] = dis[u] + edge[i].w;
                que.push( make_pair(dis[v] , v) );
            }
        }
    }
    for(int i=1; i<=n; i++) cout<<dis[i]<<" ";

    return 0;
}
```

总结

迪杰斯特拉算法堆优化后效率很高，优于SPFA算法，但是不能用于带负权边的图，时间复杂度上堆优化效率高于朴素算法，空间复杂度上邻接表法优于邻接矩阵法。
PS : 这里不体现邻接矩阵法，没意义

补充说明

再来说设计有负权值的Bellman-Ford算法和SPFA算法
SPFA 算法是 **Bellman-Ford**算法 的**队列优化算法**的别称，通常用于求含负权边的单源最短路径，以及判负权环。SPFA 最坏情况下时间复杂度和朴素 Bellman-Ford 相同，为 O(VE)。
所以说SPFA是Bellman-Ford算法的改进版，所以Bellman-Ford算法略过

四. SPFA算法

算法原理

SPFA是Shortest Path Faster Algorithm,是Bellman-Ford算法的改进版。和其他最短路算法一样，都是以松弛操作的三角形不等式为基础操作的。
若给定的图存在负权边，类似Dijkstra算法等算法便没有了用武之地，SPFA算法便派上用场了。

算法推导

不会在这里直接甩出一个算法让大家硬背。我们看看我们能不能自己推导出SPFA算法呢?
PS: 不能有**负环**

由于负权边的存在，我们不能再保证每次松弛过后的最小值是最短路，解决这个事情很简单。既然你不一定是最短路，那我就接着松弛你即可。直到你无法再松弛了，那必定是最短路了。

既然我们无法保证最小的那个是最短路了，那我们还有必要去选出最小值吗?
答案是没必要的，因为选出最小值不仅无法做到确定最值，还白白增加了时间复杂度。
所以我们不再需要优先队列存储，只需要最简单的**队列**即可。

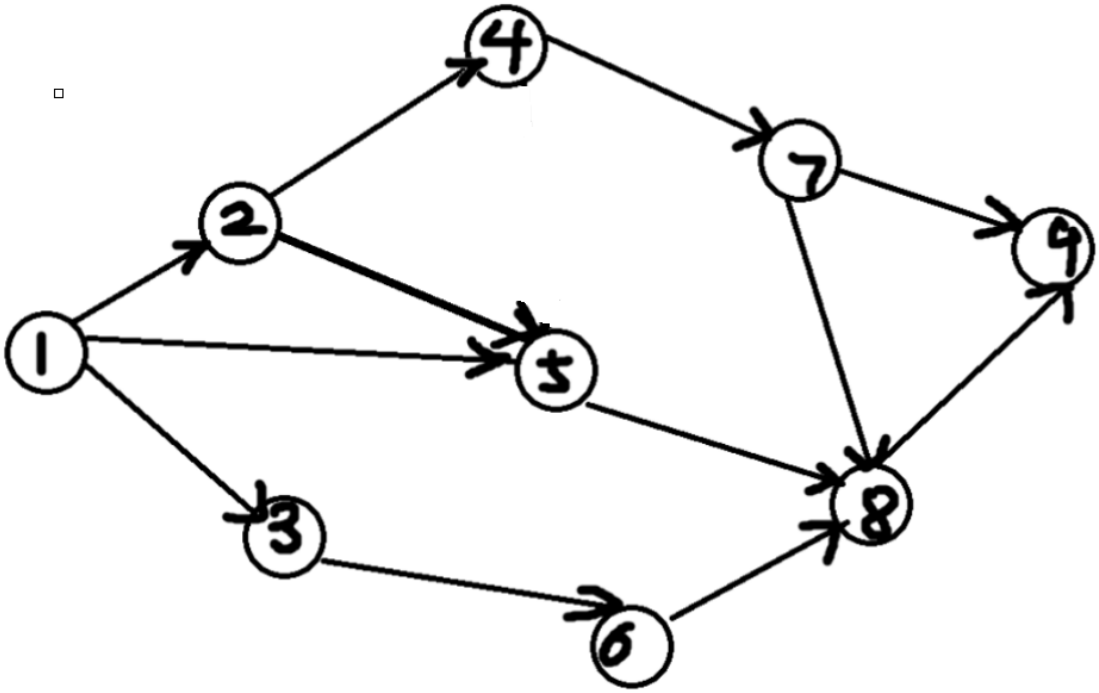
所以我们的第一个改变就是：**不再选出最小值。**

接着，那我们怎么知道所有点都无法再继续松弛了呢？此时，我们发现，我们之前在实现优化的Dijkstra算法的时候，我们只让松弛过后的点进队列。所以，如果不再入队了，那么就说明这个点无法松弛了，也就是说找到了最短路。因此，当**队列为空**的时候，就说明已经所有点的最短路都找到了。

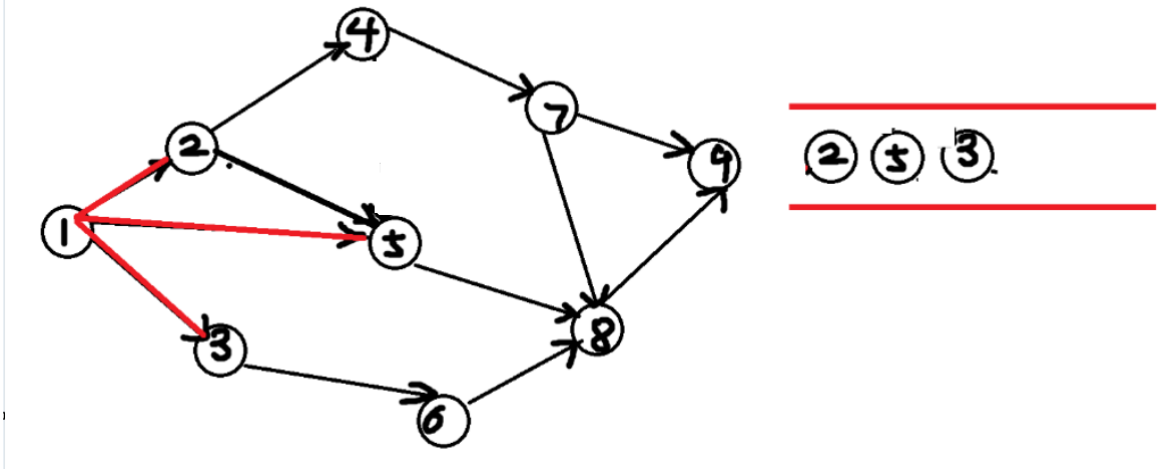
这里需要注意两个细节：

出队之后的点还能入队吗?
在我们优化Dijkstra的算法中，我们出队的是最小点，此时这个点的最短路确定了，所以不再入队。但是，现在来看，我们的点即使出队了，但依旧无法确定是最小路径，因为有负数嘛，所以你不知道这个点能够利用当前的路再去松弛，因此，出队的点依旧能够再次进队

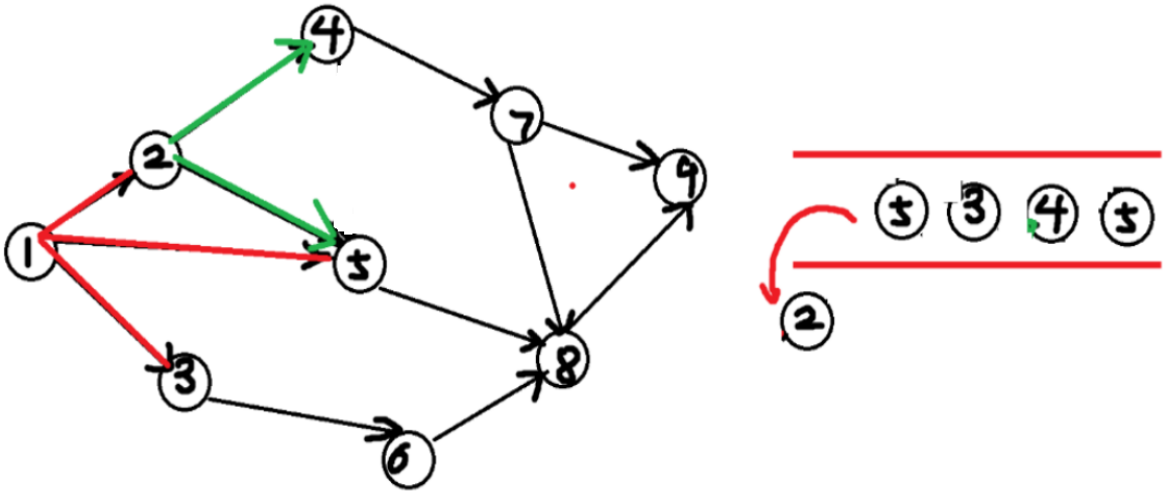
接着我们看下面这个例子，再去解决下一个细节



我们让第一个点入队，然后让这个点去松弛剩下的点，那么松弛成功的应该是1的邻接点：2，3，5
同时也说明，我们此时依旧只需要去松弛邻接点。
此时队列中的元素如下：



那么接着我们让2出队，去继续松弛。
假设我们2的邻接点都松弛成功了



那么此时的队列中，我们惊奇的发现，5进队了两次

下一个问题：**一个点有没有必要在队列中出现多次?**

松弛的原则永远是 `dis[v] > dis[u] + edge[i].w`

我们松弛过后，改变的数据是对应点的dis数组，当2出队时，利用公式松弛之后，dis[5]发生了改变。当第一个5出队的时候，此时注意我们利用的dis[5]并没有改变，然后，轮到我们第二个5出队的时候，dis[5]还是没有发生改变。即两次我们重复了。

因此，**某个点无需多个同时存在于队列中**，队列中有一个即可

最后，我们总结一下我们的优化思路：
不断的松弛邻接点，松弛成功的点入队，直到队列为空，说明最短路已经全部找到，其中有两个细节：一是出队的点可以再次入队，二是某个点只需要在队列中同时存在一个。

那么这就是SPFA的算法逻辑！！

算法分析

简洁起见，我们用数组dis记录每个结点的最短路径估计值，而且用链式前向星来存储图。
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。

举例如下，定义V0为起点，求从V0点到其他点的最短距离，用dis数组存储

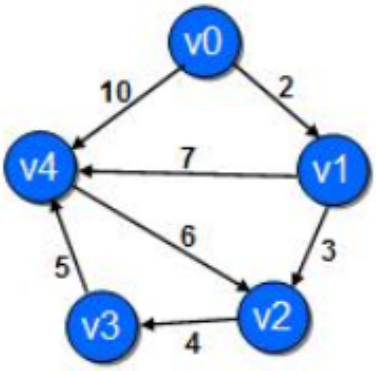


图1 有向图G，求源点v0到各点的最短路

q	v0				

	v0	v1	v2	v3	v4
dis	0	∞	∞	∞	∞

(1) 源点入队，dis[v0]=0,其余为∞

q	v1	v4			

	v0	v1	v2	v3	v4
dis	0	2	∞	∞	10

(2) 源点v0出队，v1,v4入队；
dis[v1]=2,dis[v4]=10

q	v4	v2			

	v0	v1	v2	v3	v4
dis	0	2	5	∞	9

(3) v1出队，v2入队；
dis[v2]>dis[v1]+a[v1][v2]=5,更新
dis[v4]>dis[v1]+a[v1][v4]=9,更新

q	v2				

	v0	v1	v2	v3	v4
dis	0	2	5	∞	9

(4) v4出队

q	v3				

	v0	v1	v2	v3	v4
dis	0	2	5	9	9

(5) v2出队，v3入队
dis[v3]>dis[v2]+a[v2][v3]=9,更新

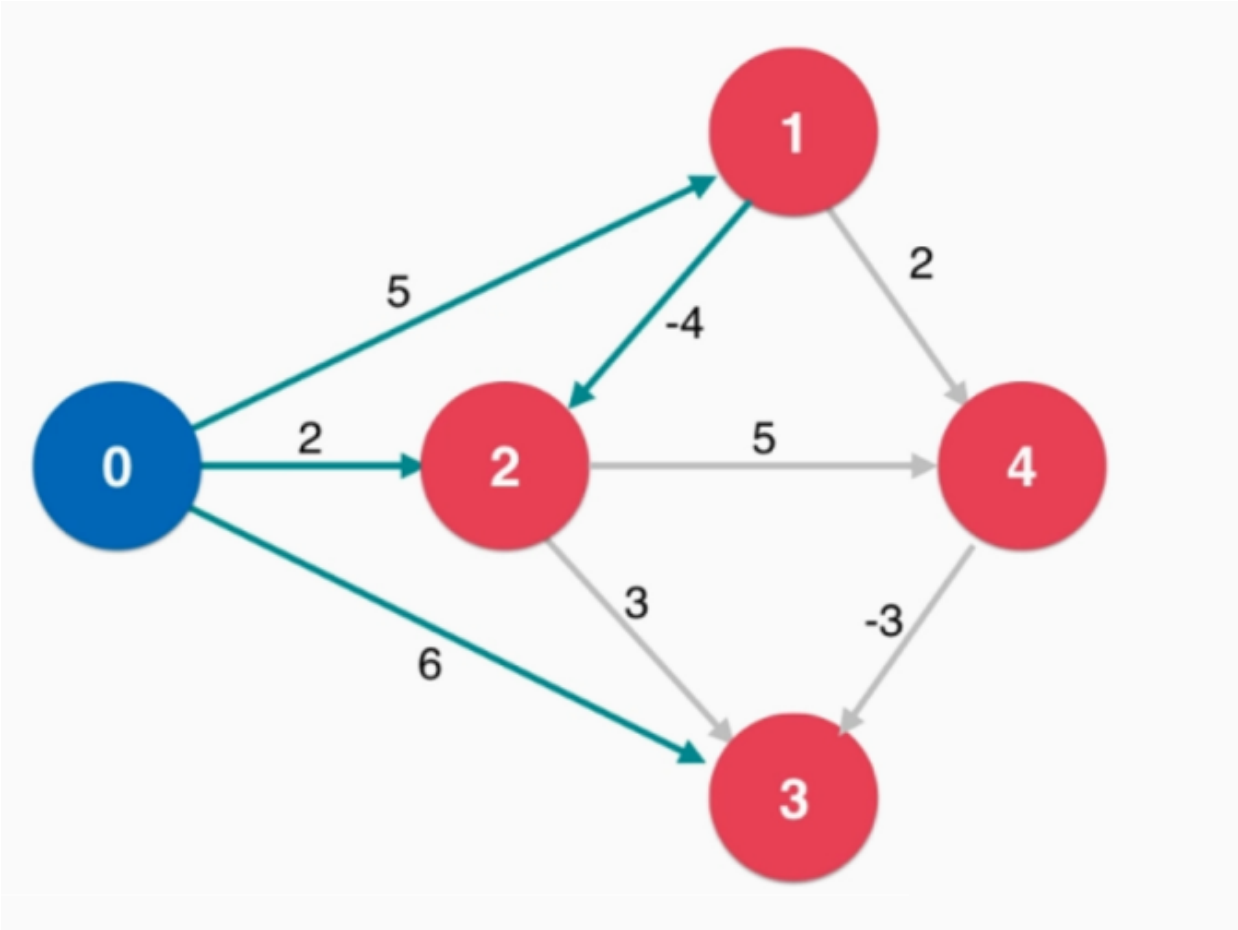
q	v3				

	v0	v1	v2	v3	v4
dis	0	2	5	9	9

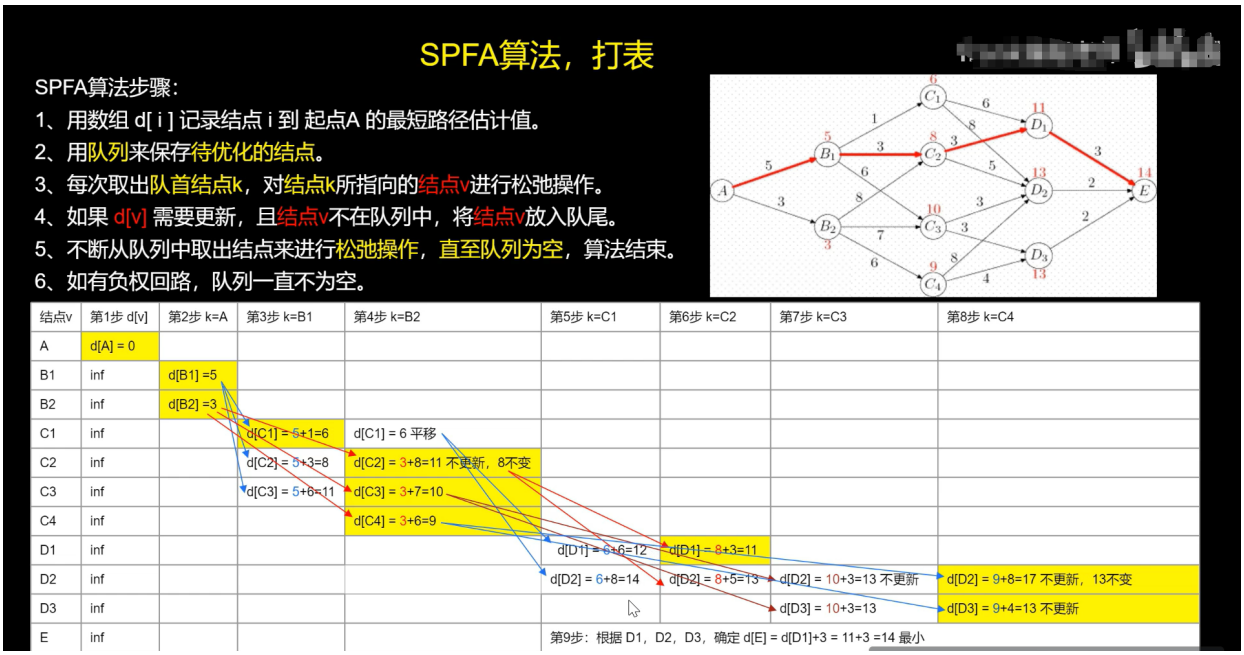
(6) v3出队，队列空
V0到各点的最短距离已求出

图2 spfa算法的执行步骤及队列、dis数组变化情况

按照上述方法请推导下图：



每次将点放入队尾，都是经过**松弛操作**达到的。换言之，每次的优化将会有某个点v的最短路径估计值dis[v]变小。所以算法的执行会使d越来越小。算法不会无限执行下去，随着d值的逐渐变小，直到到达最短路径值时，算法结束，这时的最短路径估计值就是对应结点的最短路径值。



1.用dis数组记录点到有向图的任意一点距离，初始化起点距离为0，其余点均为INF，起点入队。

2.判断该点是否存在。（未存在就入队，标记）

3.队首出队，并将该点标记为没有访问过，方便下次入队。

4.遍历以对首为起点的有向边（t,i）,如果dis[i]>dis[t]+w(t,i),则更新dis[i]。

5.如果i不在队列中，则入队标记，一直到循环为空。

```
#include<bits/stdc++.h>
using namespace std;
#define int long long
struct node{
    int to,w,next;
}edge[10005];
int first[10005] , cnt , n , m , u , v , w;
int dis[10005] , visit[10005];
queue<int> que;
void add(int p,int q,int r){
    edge[++cnt].to = q;
    edge[cnt].w = r;
    edge[cnt].next = first[p];
    first[p] = cnt;
}
signed main(){

    cin >> n >> m;
    for(int i=1;i<=m;i++){
        cin >> u >> v >> w;
        add(u,v,w); // 有向图
    }
    //SPFA
    for(int i=1;i<=n;i++) dis[i]=INT_MAX;
    // 从1号点开始
    dis[1] = 0;
    visit[1] = 1;
    que.push(1);
    while( !que.empty() ){
        int x = que.front();
        que.pop();
        visit[x] = 0;
        for(int i=first[x] ; i!=0 ; i=edge[i].next){
            int y = edge[i].to;
            if( dis[y] > dis[x] + edge[i].w ){
                dis[y] = dis[x] + edge[i].w;
                if(visit[y]==1) continue;
                que.push(y);
                visit[y]=1;
            }
        }
    }
    for(int i=1;i<=n;i++){
        if(dis[i]!=INT_MAX){
            cout << dis[i] << " ";
```

```
        }else{
            cout << -1 << " "; // 代表不连通
        }
    }

    return 0;
}
```

算法SLF优化

SLF优化，即Small Label First策略，使用STL中的双端队列deque容器来实现，较为常用。
还是在贪心的基础上，再做松弛操作时，先优先处理dis值较小的那个点，即在原有的SPFA算法中**每次判断扩展出的点与队首元素进行判断。**即：**对要加入队列的点 u，如果 dis[u] 小于队首元素 v 的 dist[v]，将其插入到队头，否则插入到队尾。**这样就能减少一些浪费时间的遍历操作。

注：队列为空时直接插入队尾。

```
// 从s到其他城镇的单源最短路径
for(int i=1;i<=t;i++) dis[i] = INT_MAX;
dis[s] = 0;
visit[s] = 1;
que.push_back(s);
while( !que.empty() ){
    int x = que.front();
    que.pop_front();
    visit[x] = 0;
    for(int i=first[x] ; i!=0 ; i=edge[i].next){
        int y = edge[i].to;
        if( dis[y] > dis[x] + edge[i].w ){
            dis[y] = dis[x] + edge[i].w;
            if( visit[y]==0 ){
                visit[y] = 1;
                if( !que.empty() and dis[y] < dis[que.front()] ){
                    que.push_front(y);
                }else{
                    que.push_back(y);
                }
            }
        }
    }
}
```

小细节注意

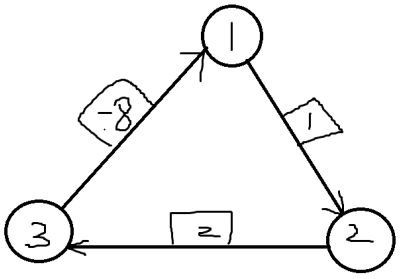
虽然大多数情况SPFA跑的比较快，但时间复杂度仍为(Onm)，主要用应用于有负边权的情况（如果没有负边权，推荐使用**Dijkstra**算法），但该算法很容易超时被卡，必须要谨慎选择该算法，如果是正权图，SPFA算法有风险，优先还是**Dijkstra**算法。

在有负环的情况下，不存在最短路，因为不停在负环上绕就能把最短路刷到任意低。但是SPFA能够判断图中是否存在负环，具体方法为统计每个点的入队次数，如果有一个点入队次数为n，那么图上存在负环，也就不存在最短路了。

其他应用之判断负环

①什么是负环

负环，又叫负权回路，负权环，指的是一个图中存在一个环，里面包含的边的边权总和<0。
在存在负环的图中，是求不出最短路径的，因为只要在这个环上不停的兜圈子，最短路径就会无限小,如下图：



②如何判断负环

判断负权回路的方法有两种
1、如果一个点的入队的次数大于等于点的总数N，则存在负权回路。
2、如果每个顶点的最短路径超过N，则存在负权回路。

重点来说第一种，其实非常好理解，代码如下：
用num数组记录每一个顶点的入队次数，如果入队此时>=n ，即存在负环

```
#include<bits/stdc++.h>
using namespace std;
#define int long long
int T , n , m , u , v , w , first[2005] , cnt , num[2005] , visit[2005] , f;
int dis[2005];
struct node {
    int to,next,w;
} edge[3005*2];
inline void add(int u,int v,int w) {
    cnt++;
    edge[cnt].to = v;
    edge[cnt].w = w;
    edge[cnt].next = first[u];
```

```
    first[u] = cnt;
}
queue<int> que;
inline void spfa() {
    que.push(1);
    visit[1] = 1;
    num[1] = 1;
    dis[1] = 0;
    while( !que.empty() ) {
        int x = que.front();
        que.pop();
        visit[x] = 0;
        for(int i=first[x] ; i ; i=edge[i].next) {
            int y = edge[i].to;
            if( dis[y] > dis[x] + edge[i].w ) {
                dis[y] = dis[x] + edge[i].w;
                if( visit[y]==0 ) {
                    visit[y] = 1;
                    que.push(y);
                    num[y]++;
                    if(num[y]>n){
                        cout << "YES\n";
                        return;
                    }
                }
            }
        }
    }
    cout<<"NO\n";
    return;
}
signed main() {

    cin >> T;
    while(T--) {
        cin >> n >> m;
        cnt = 0;
        memset(first,0,sizeof(first));
        memset(num,0,sizeof(num));
        memset(visit,0,sizeof(visit));
        while( !que.empty() ) que.pop();
        for(int i=1; i<=n; i++) dis[i]=INT_MAX;
        while(m--) {
            cin >> u >> v >> w;
            if(w>=0) {
                add(u,v,w);
                add(v,u,w);
            } else {
                add(u,v,w);
            }
        }
        spfa();
```

```
    }


    return 0;
}
```

总结

SPFA的时间复杂度是O(km)是每一个节点的平均入队次数，经过实践，k一般为4，所以SPFA通常情况下非常高效。但是SPFA非常容易被卡出翔，最坏情况下会变成O(nm)，所以如果能**用Dijkstra尽量不要用SPFA**，尤其是正权图，不要偷懒。