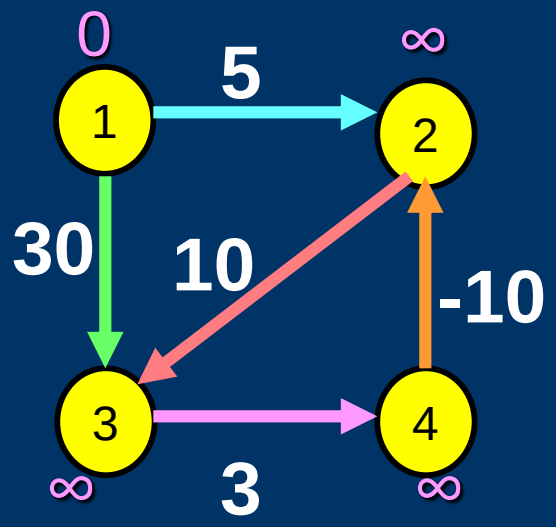
图论3 # SPFA

Shortest Path Faster Algorithm

复习 Bellman-ford算法



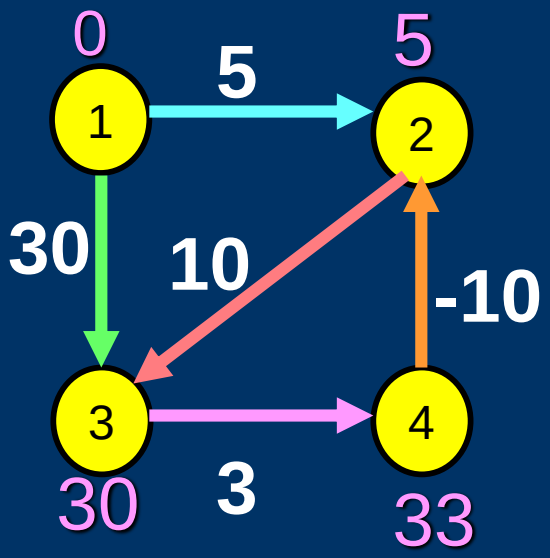
dis[2]= ∞ = dis[4]+line[4][2]= ∞ dis[2]= ∞

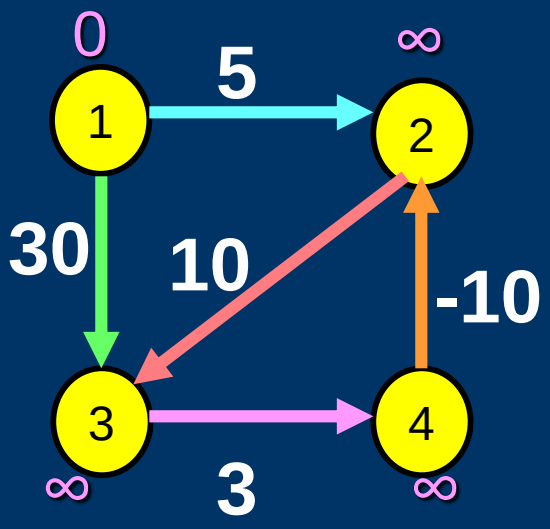
dis[3]= ∞ = dis[2]+line[2][3]= ∞ dis[3]= ∞

dis[3]=∞ > dis[1]+line[1][3]=30 dis[3]=30

dis[4]=∞ > dis[3]+line[3][4]=33 dis[4]=33

dis[2]=∞ > dis[1]+line[1][2]=5 dis[2]=5





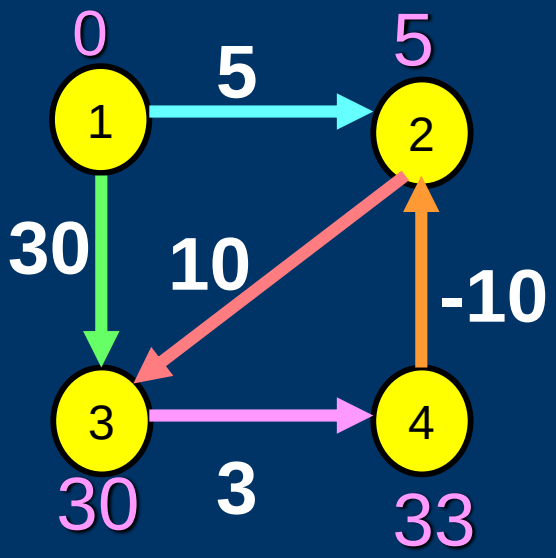
dis[2]= 5 < dis[4]+line[4][2]= 23 dis[2]= 5

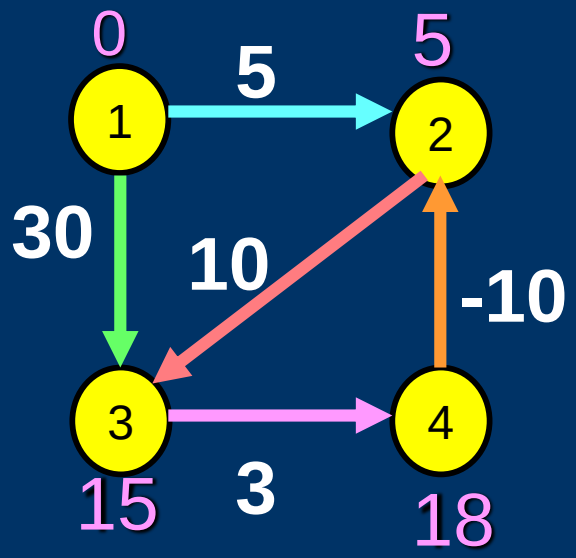
dis[3]= 30 > dis[2]+line[2][3]= 15 dis[3]= 15

dis[3]=15 < dis[1]+line[1][3]=30 dis[3]=15

dis[4]=33 > dis[3]+line[3][4]=18 dis[4]=18

dis[2]=5 = dis[1]+line[1][2]=5 dis[2]=5





```
struct bian{
            int x,y,w;
          };
   bian line[maxm];
   int dis[maxn],path[maxn];
   boo f;
```

line[i].x表示边i的起点
line[i].y表示边i的终点
line[i].w表示边i的长度

//dis存每个点到起点的距离
//path记录路径上前一个点的编号
//f记录是否有负权回路

```
void bellman()
{
   int i,j;
   for(i=1;i<=n;i++)dis[i]=INT_MAX;
   dis[s]=0;

  for(i=1; i<=n;i++)
    for(j=1;j<=m;j++)
      if(dis[line[j].x]+line[j].w<dis[line[j].y])
      {
        dis[line[j].y]= dis[line[j].x]+line[j].w;
        path[line[j].y]=line[j].x;
      }
//判断是否存在负权回路
  for(j=1;j<=m;j++)
    if(dis[line[j].x]+line[j].w<dis[line[j].y])
    f=true;
}
```

//赋初值，将每个点到起点的距离设为无限大
//起点s到本身的距离设置为0

//进行n趟松弛
//每趟对所有边进行松弛操作

//如果点y经过边j后到起点的距离缩短，更新距离

//如果没有负权回路，那么每个点到起点的最短距离是固定的。在对每条边松弛一次，如果发现有点到起点的距离缩短，说明该图存在负权回路

Bellman-ford算法特点

1.时间复杂度　O(nm)　n 表示节点数，m表示边数

2.可用于负权（但不能有负权回路）

更深入的分析：

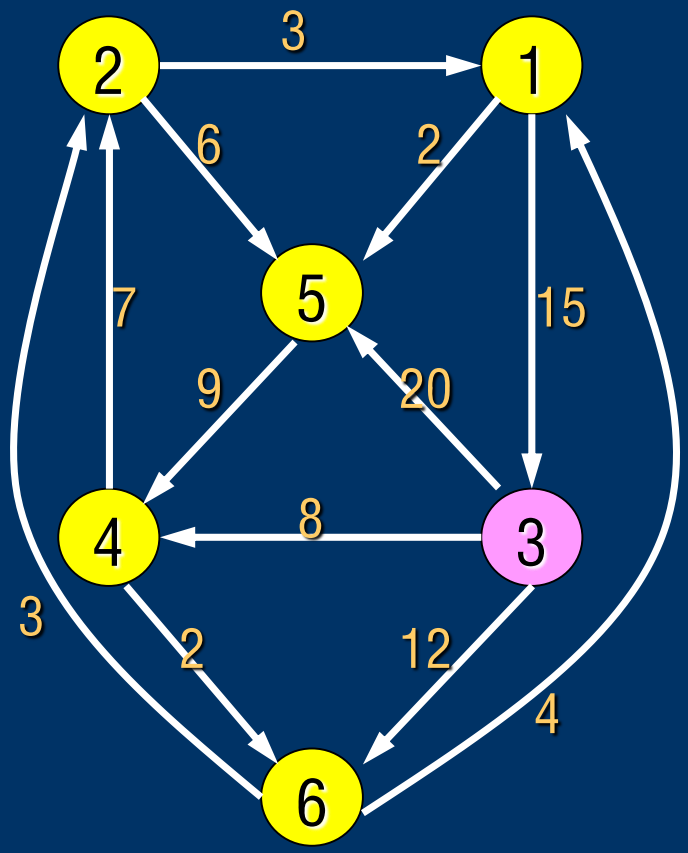
Bellman ford是对所有边进行n趟松弛操作，不管这趟松弛是否有点到起点的距离发生变化。

但实际上在松弛过程中如果有点到起点的距离缩短才有可能导致其他点到起点的距离变化。

所以，如果只把距离缩短的点相关的边进行松弛，距离没有发生变化的点的边则不变化，那么就能减少松弛操作的次数。

改进：在处理的时候把松弛过程中到起点距离缩短的点放到一个队列中，只对该队列中的点进行松弛操作。

SPFA

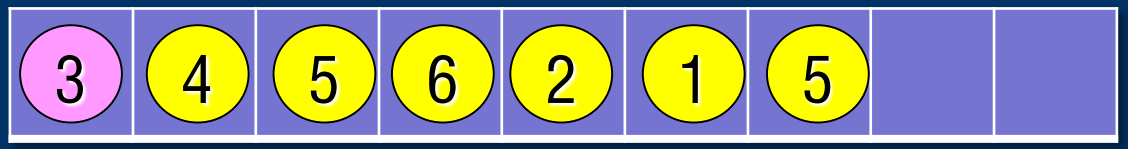


地图上有6座城市（编号1到6），它们通过很多单行道连接，请找出从3号城市出发到其他所有城市的最短距离

下面数组存每个点到起点（3号点）的最短距离：

3	1	2	4	5	6
dis	0	14	13	8	16	10

队列：



SPFA

SPFA是Bellman-Ford算法的一种队列实现，减少了不必要的冗余计算。

算法流程

用一个队列来进行维护。初始时将起点加入队列。每次从队列中取出一个元素，并对所有与他相邻的点进行松弛，若某个相邻的点松弛成功（到起点距离缩短），则将其入队。直到队列为空时算法结束。

简单的说就是队列优化的bellman-ford,利用了每个点不会更新次数太多的特点

SPFA的时间复杂度是O(kE) k一般取2左右（k是增长很快的函数ackermann的反函数,2^65536次方也就5以下 ），可以处理负边。

SPFA的实现甚至比Dijkstra或者 Bellman_Ford还要简单。

```
queue<int>q;
int dis[maxn];                    // q为队列,dis记录节点到起点的距离
bool f[maxn];                     //用于标记节点是否在队列中

void spfa(int s)                  //s为起点,求s到图中所有点的距离
{
    int i,x,
    for(i=1;i<=n;i++)dis[i]=99999999;     //将距离初始化为一个很大的值
    q.push(s);     f[s]=true;     dis[s]=0;      //初始化起点，将其入队

    while(q.size()>0)
    {
        x=q.front();                          //x存队首元素的编号
        q.pop();          f[x]=false;         //队首元素出队，将其标记为不在队中
        for(i=1;i<=n;i++)                     //讨论n个点中与x相连的点
            if (dis[x]+map[x][i]<dis[i])      //若经过x号点，起点到i的距离缩短，则更新距离
            {
                dis[i]=dis[x]+map[x][i];
                if (f[i]==false)              //如果i点松弛成功且不在队中，入队
                {
                    q.push(i);     f[i]=true;   //i号点进队，标记为已在队列中
                }
            }
     }
}
```

SPFA手工队列版

```
void spfa(int s)                          //s为起点
 {
      int q[4*maxn],dis[maxn];      // q为队列,dis记录节点到起点的距离
      bool f[maxn];                       //用于标记节点是否在队列中
      int i,head,tail,x;

      for(i=1;i<=n;i++)dis[i]=99999999;    //将距离初始化为一个很大的值
      q[1]=s;     f[s]=true;   dis[s]=0;         //初始化起点，将其入队
      head=1;   tail=2;

      while(head!=tail)
     {
           x=q[head++];  f[x]=false;      //队首元素出对，将其标记为不在队中
           for(i=1;i<=n;i++)                  //讨论n个点中与x相连的点
              if (dis[x]+map[x][i]<dis[i])
            {
                  dis[i]=dis[x]+map[x][i];
                  if (f[i]==false)                //如果i点松弛成功且不在队中，入队
                  {
                        q[tail++]=i;     f[i]=true;
                  }
              }
       }
}
```

用SPFA判定负权回路

想一想：

在SPFA算法中，每个点最多进队多少次?

N-1次，因为对于一个点x,最坏情况下是其余N-1个点都可以将其松弛。

所以，若一个点进队次数超过了N次，我们就可以认定，该图中存在负权回路，可以果断结束SPFA了。

```
queue<int>q;
int dis[maxn];                    // q为队列,dis记录节点到起点的距离
bool f[maxn];                     //用于标记节点是否在队列中
int cnt[maxn];                    //申明一个数组，用于统计每个点进队的次数

void spfa(int s)                  //s为起点,求s到图中所有点的距离
{
    int i,x,
    for(i=1;i<=n;i++)dis[i]=99999999;     //将距离初始化为一个很大的值
    q.push(s);     f[s]=true;     dis[s]=0;      //初始化起点，将其入队

    while(q.size()>0)
    {
        x=q.front();                          //x存队首元素的编号
        q.pop();          f[x]=false;         //队首元素出队，将其标记为不在队中
        for(i=1;i<=n;i++)                     //讨论n个点中与x相连的点
            if (dis[x]+map[x][i]<dis[i])      //若经过x号点，起点到i的距离缩短，则更新距离
            {
                dis[i]=dis[x]+map[x][i];
                if (f[i]==false)              //如果i点松弛成功且不在队中，入队
                {
                    q.push(i);    f[i]=true;  //i号点进队，标记为已在队列中
                    cnt[i]++;                 //记录i号点进队的次数
                    if(cnt[i]==n){  cout<<"有负权环";  return;  }
                }
            }
    }
}
```

SPFA手工队列版

```
void spfa(int s)                          //s为起点
 {
      int q[4*maxn],dis[maxn];      // q为队列,dis记录节点到起点的距离
      bool f[maxn];                       //用于标记节点是否在队列中
      int i,head,tail,x;
      int cnt[maxn];                 //申明一个数组，用于统计每个点进队的次数

      for(i=1;i<=n;i++){  dis[i]=99999999; cnt[i]=0;   }
      q[1]=s;     f[s]=true;   dis[s]=0;    cnt[s]=1;     //初始化起点，将其入队
      head=1;   tail=2;

      while(head!=tail)
     {
           x=q[head++];  f[x]=false;      //队首元素出对，将其标记为不在队中
           for(i=1;i<=n;i++)                  //讨论n个点中与x相连的点
              if (dis[x]+map[x][i]<dis[i])
            {
                   dis[i]=dis[x]+map[x][i];
                   if (f[i]==false)                 //如果i点松弛成功且不在队中，入队
                   {
                         q[tail++]=i;     f[i]=true;
                         cnt[i]++;
                         if(cnt[i]==n){cout<<"有负权环";  return;  }
                    }
              }
       }
}
```

spfa 小结

1. 时间复杂度O(kE)

   k指每个点的平均进队次数，一般为2

   E指边的总数，所以期望时间复杂度为O(2E)

2. 可用于负权

3. 可判断负权回路

   每个点的进队次数不超过N

最长路怎么求?

怎样卡掉SPFA?

- 1560 1976 2225 2226