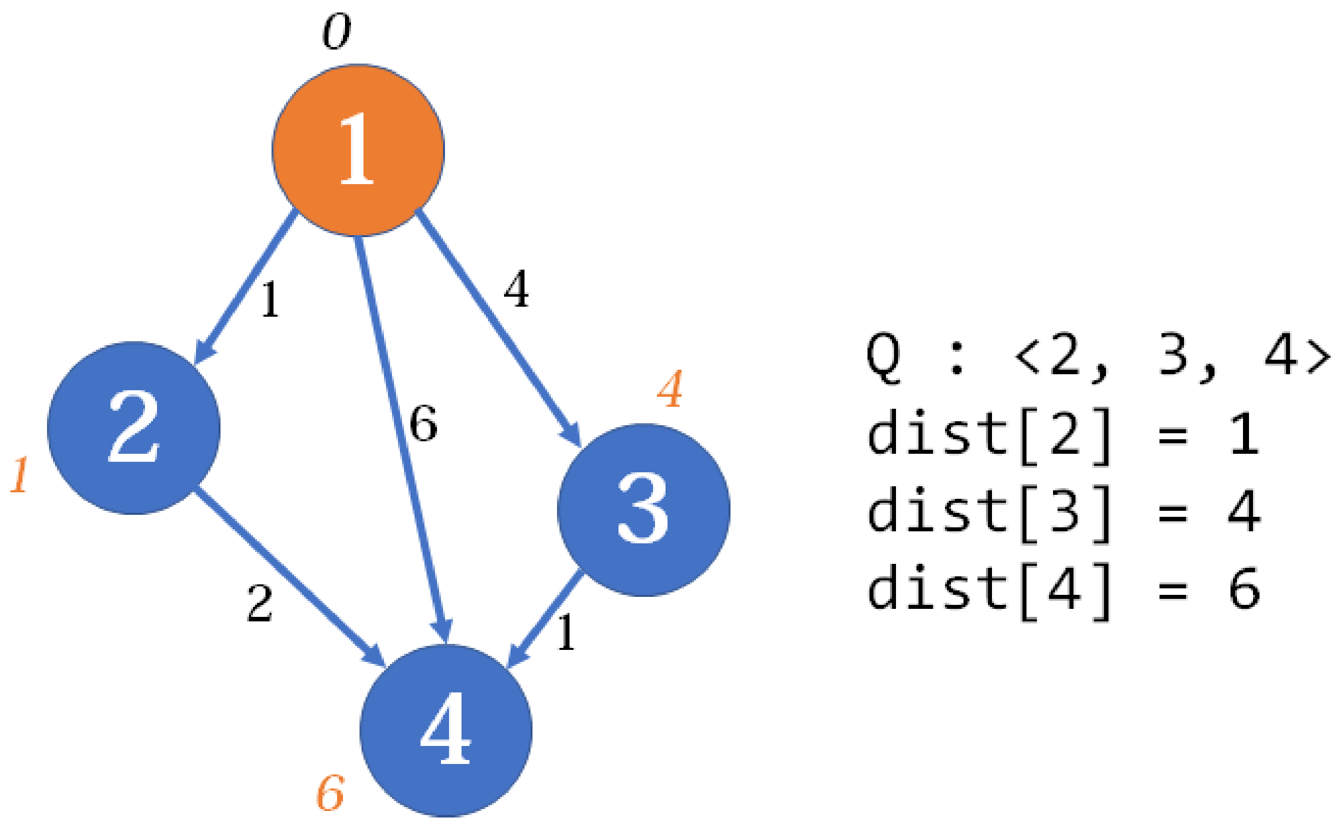
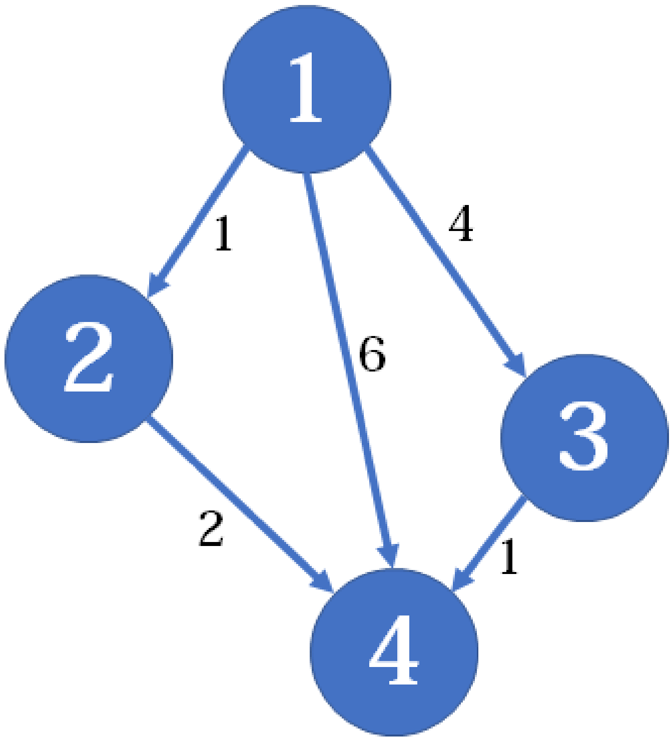


最短路问题

这篇文章应该会很长，因为我们要探讨图论中一个基本而重要的问题：**最短路问题**。如下图，我们想知道，**某点到某点最短的路径有多长**？



图中点1到点4的最短路径长度应为3

最短路问题分为两类：**单源最短路**和**多源最短路**。前者只需要求**一个固定的起点**到各个顶点的最短路径，后者则要求得出**任意两个顶点**之间的最短路径。我们先来看多源最短路问题。

Floyd算法

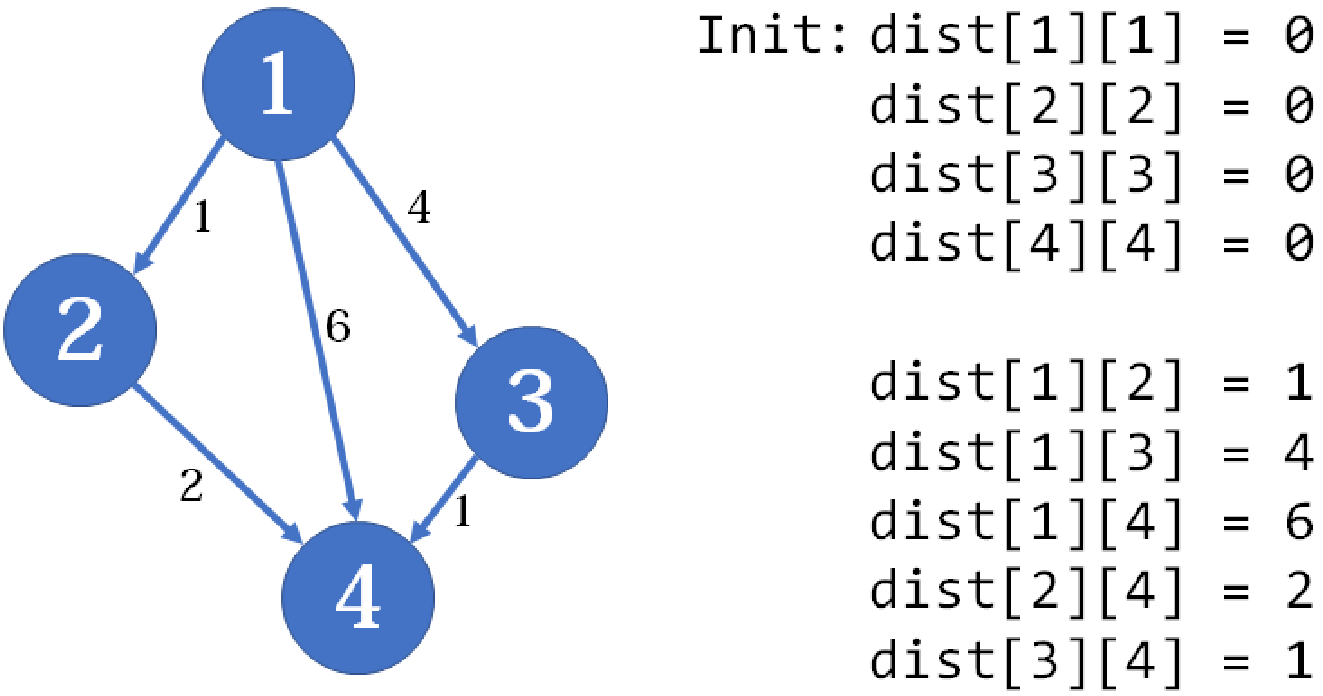
我们用Floyd算法解决多源最短路问题：

```
int dist[400][400];
void Floyd(int n)
{
    for (int k = 1; k <= n; ++k)
        for (int i = 1; i <= n; ++i)
            for (int j = 1; j <= n; ++j)
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
}
```

四行代码，简洁明了。Floyd本质上是一个**动态规划**的思想，每一次循环更新**经过前k个节点，i到j的最短路径。**

这甚至不需要特意存图，因为dist数组本身就可以从邻接矩阵拓展而来。初始化的时候，我们把每个点**到自己的距离**设为0，每新增一条边，就把从这条边的起点到终点的距离设为此边的**边权**（类似于邻接矩阵）。其他距离初始化为**INF**（一个超过边权数据范围的大整数，注意防止溢出）。

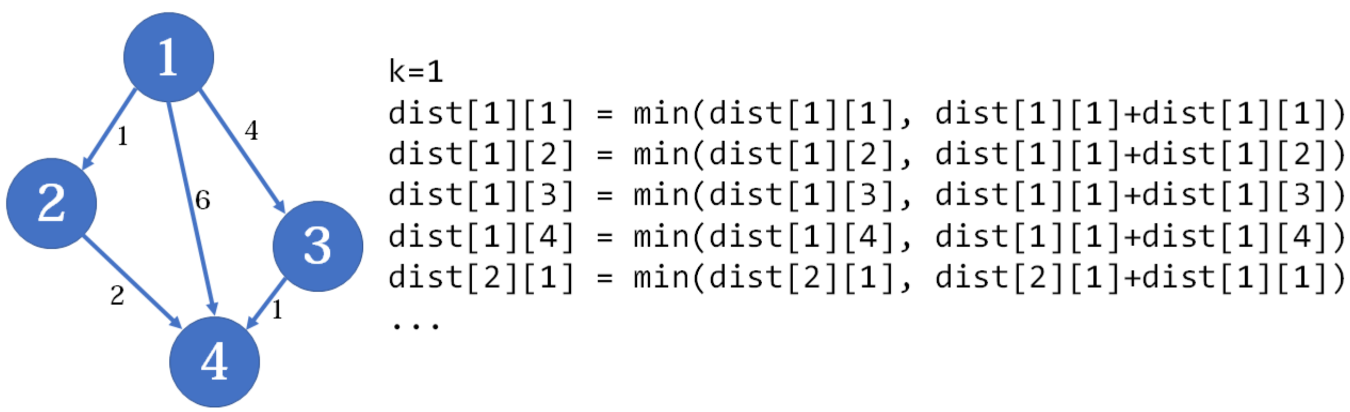
```
//Floyd初始化
memset(dist, 63, sizeof(dist));
//利用memset的特性，先把所有距离初始化为0x3f3f3f3f，注意这个数的两倍小于32位和64位机器上的INT_MAX
for (int i = 1; i <= n; ++i)
    dist[i][i] = 0;
for (int i = 0; i < m; ++i)
{
    int u, v, w;
    scanf("%d%d%d", &u, &v, &w);
    dist[u][v] = w;
}
```



初始化

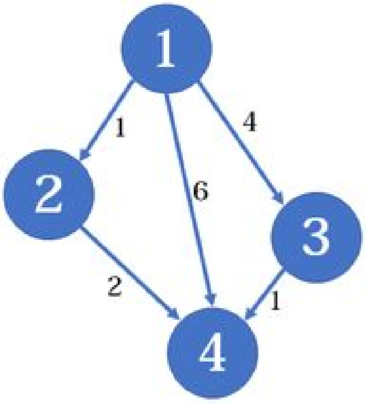
如果你还是没懂，现在我们来看Floyd的具体过程。

第一趟，k=1：



很明显，没有一个距离能通过经由1号点而减短。

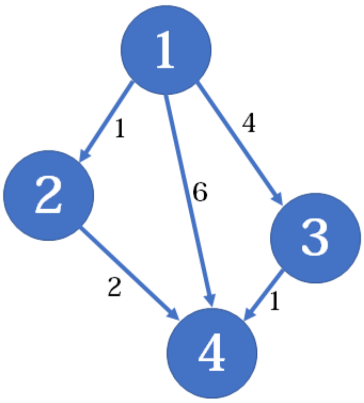
第二趟，k=2：




```
k=2
dist[1][1] = min(dist[1][1], dist[1][2]+dist[2][1])
dist[1][2] = min(dist[1][2], dist[1][2]+dist[2][2])
dist[1][3] = min(dist[1][3], dist[1][2]+dist[2][3])
dist[1][4] = min(dist[1][4], dist[1][2]+dist[2][4])
           = 3
dist[2][1] = min(dist[2][1], dist[2][2]+dist[2][1])
...
```

这里，dist[1][4]通过经由2号点，最短路径缩短了。

第三趟，k=3：




```
k=3
dist[1][1] = min(dist[1][1], dist[1][3]+dist[3][1])
dist[1][2] = min(dist[1][2], dist[1][3]+dist[3][2])
dist[1][3] = min(dist[1][3], dist[1][3]+dist[3][3])
dist[1][4] = min(dist[1][4], dist[1][3]+dist[3][4])
dist[2][1] = min(dist[2][1], dist[2][3]+dist[3][1])
...
```

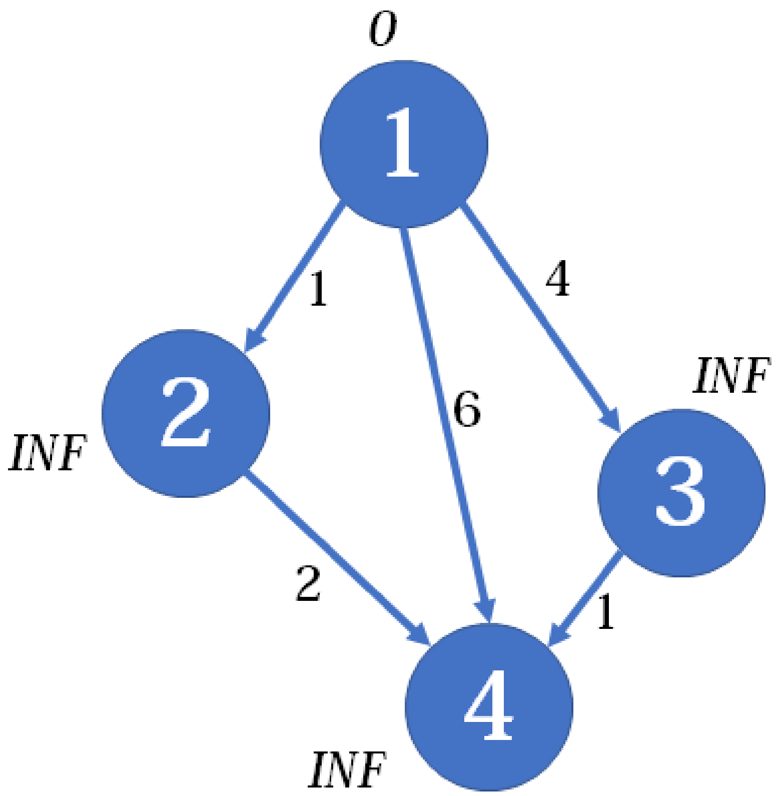
这时虽然1->3->4的路径比1->4短，但是dist[1][4]已经被更新为3了，所以这一趟又白跑了。接下来k=4显然也更新不了任何点。综上，每一趟二重循环，实际上都是在考察，**能不能经由k点，把i到j的距离缩短**？

Floyd的时间复杂度显然是 $O(n^3)$ ，同时拥有 $O(n^2)$ 的空间复杂度（本文用n表示点数，m表示边数），都比较高，所以只适用于数据规模较小的情形。

一般而言，我们更关心的是**单源最短路**问题，因为当起点被固定下来后，我们可以使用更快的算法。

Bellman-Ford算法

因为起点被固定了，我们现在只需要一个一维数组dist[]来存储每个点到起点的距离。如下图，1为起点，我们初始化时把dist[1]初始化为0，其他初始化为INF。



想想看，我们要找到从起点到某个点的最短路，设起点为S，终点为D，那这条最短路一定是**S->P1->P2->...->D**的形式，假设**没有负权环**，那这条路径上的点的总个数一定**不大于n。**

现在我们定义对点x, y的**松弛**操作是：

```
dist[y] = min(dist[y], dist[x] + e[x][y]);//这里的e[x][y]表示x、y之间的距离，具体形式可能根据存图方法不同而改变
```

松弛操作就相当于考察能否**经由x点**使**起点到y点**的距离变短。

所以要找到最短路，我们只需要进行以下步骤：

先松弛S, P1，此时dist[P1]必然等于e[S][P1]。
再松弛P1, P2，因为S->P1->P2是最短路的一部分，**最短路的子路也是最短路**（这是显然的），所以dist[P2]不可能小于dist[P1]+e[P1][P2]，因此它会被更新为dist[P1]+e[P1][P2]，即e[S][P1]+e[P1][P2]。
再松弛P2, P3，......以此类推，最终dist[D]必然等于e[S][P1]+e[P1][P2]+...，这恰好就是最短路径。

说得好像很有道理，但是问题来了，我怎么知道这些P1、P2是什么呢？我们不就是要找它们吗？关键的来了，Bellman-Ford算法告诉我们：

把所有边松弛一遍！

因为我们要求的是最小值，而多余的松弛操作不会使某个dist比最小值还小。所以**多余的松弛操作不会影响结果**。把所有边的端点松弛完一遍后，我们可以保证S, P1已经被松弛过了，现在我们要松弛P1, P2，怎么做呢？

再把所有边松弛一遍！

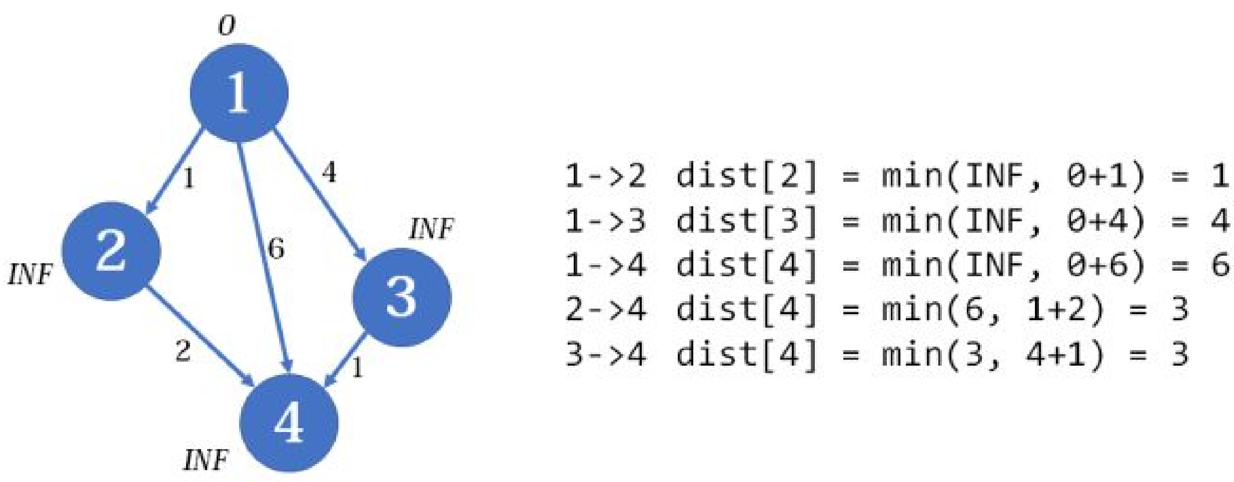
好了，现在我们松弛了P1, P2，继续这么松弛下去，什么时候是尽头呢？还记得我们说过吗？最短路上的点的总个数一定**不大于n**，尽管一般而言最短路上的顶点数比n少得多，但反正多余的松弛操作不会影响结果，我们索性：

把所有边松弛n-1遍！

这就是Bellman-Ford算法，相信你已经意识到，这是种很暴力的算法，它的时间复杂度是 $O(nm)$ 。代码如下：

```
void Bellman_Ford(int n, int m)
{
    for (int j = 0; j < n - 1; ++j)
        for (int i = 1; i <= m; ++i)
           dist[edges[i].to] = min(dist[edges[i].to], dist[edges[i].from] + edges[i].w);
}
```

三行代码，比Floyd还简单。这里用的是链式前向星存图，但是建议存的时候多存一个from，方便遍历所有边。当然其实并没什么必要，这里直接**暴力存边集**就可以了，因为这个算法并不关心每个点能连上哪些边。



很显然我这个图太简单了一点，只遍历了一遍所有边，就把所有最短路求出来了。但为了保证求出正解，还需要遍历两次。

我们之前说，我们不考虑**负权环**，但其实Bellman-Ford算法是可以很简单地处理负权环的，只需要再**多对每条边松弛一遍**，如果这次还有点被更新，就说明存在负权环。因为没有负权环时，最短

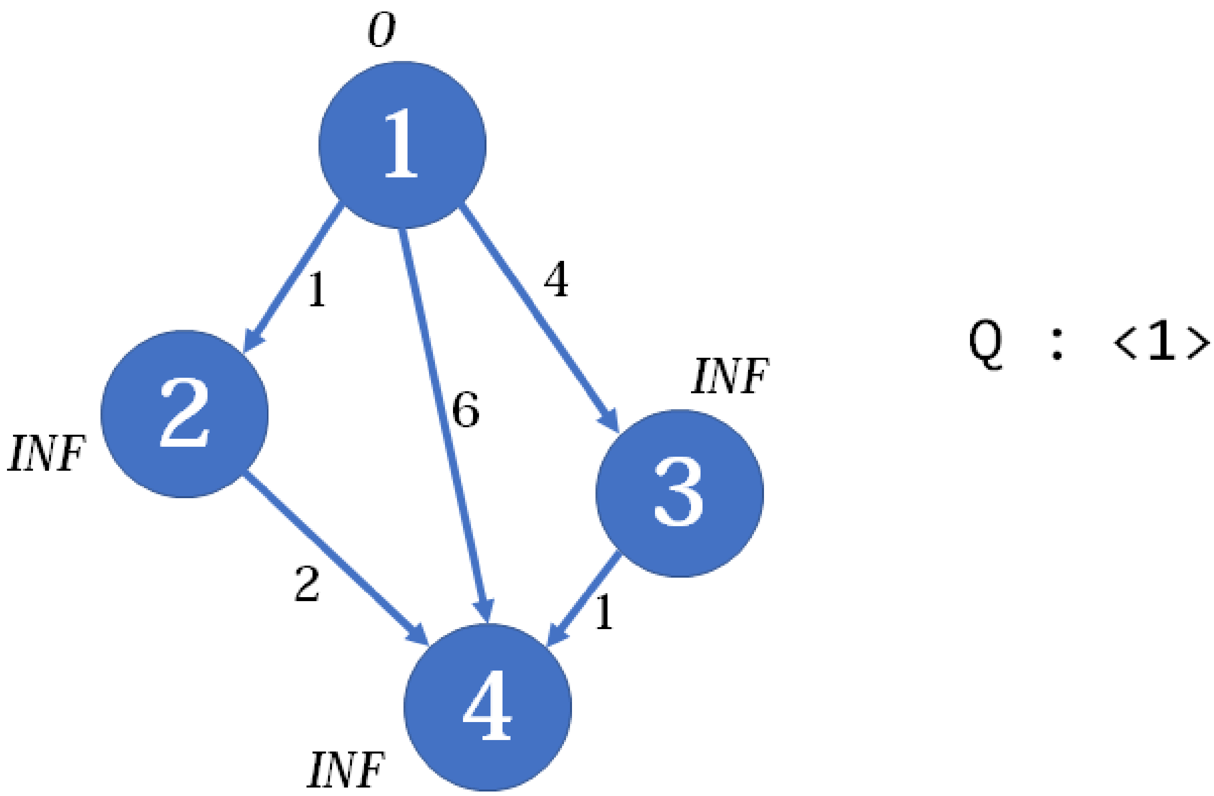
路上的顶点数一定小于n，而存在负权环时，可以无数次地环绕这个环，最短路上的顶点数是无限的。

SPFA算法

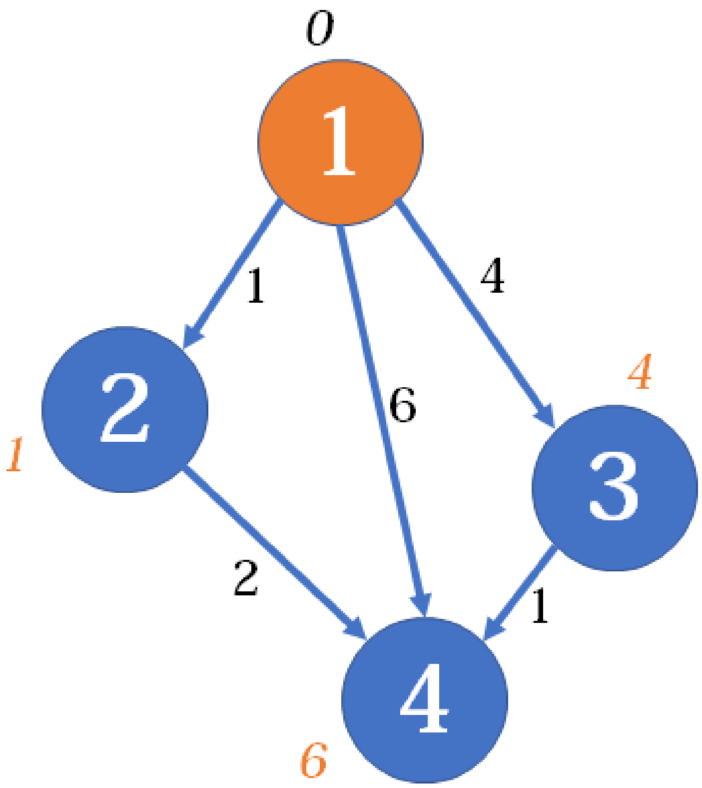
$O(nm)$ 的复杂度显然还是太高了，现在我们想想，能不能别这么暴力，每次不松弛所有点，而只松弛**可能更新**的点？

我们观察发现，第一次松弛S, P1时，可能更新的点只可能是**S能直接到达的点**。然后下一次可能被更新的则是**S能直接到达的点能直接到达的点**（禁止套娃？）。SPFA算法正是利用了这种思想。

SPFA算法，也就是**队列优化**的Bellman-Ford算法，维护一个队列。一开始，把起点放进队列：

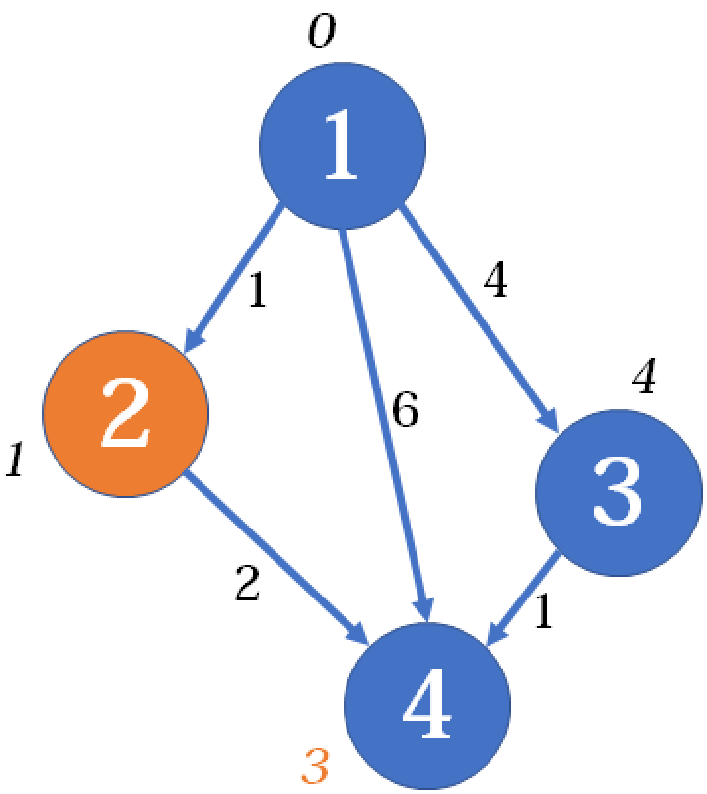


我们现在考察1号点，它可以到达点2、3、4。于是1号点出队，2、3、4号点依次入队，入队时松弛相应的边。




```
Q : <2, 3, 4>
dist[2] = 1
dist[3] = 4
dist[4] = 6
```

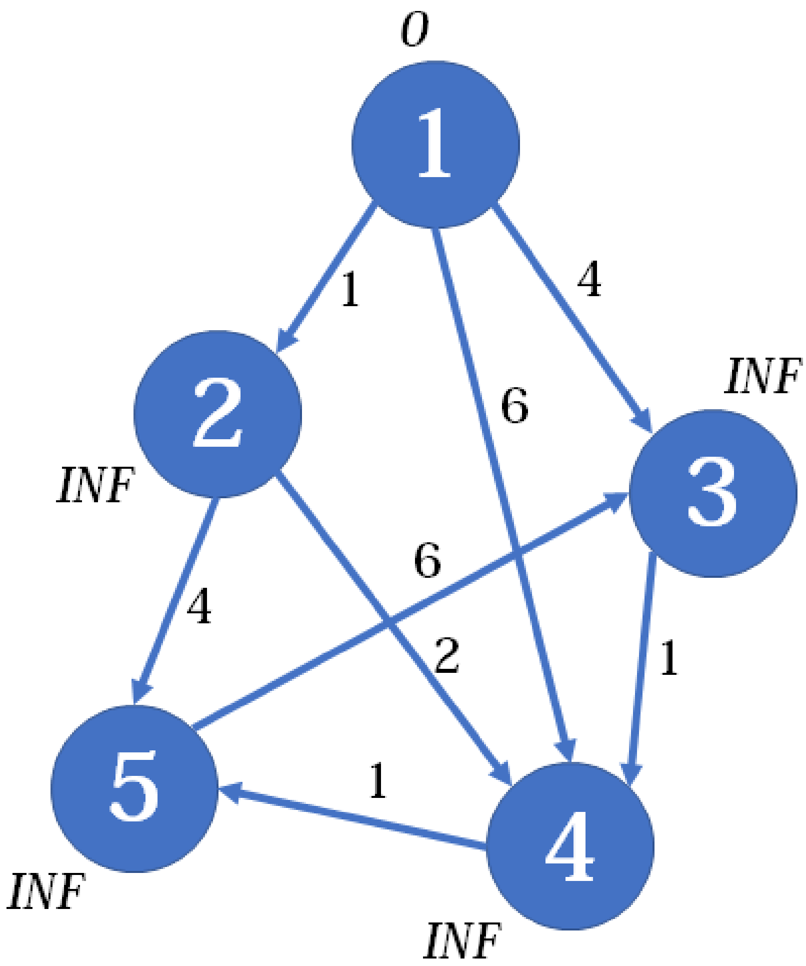
现在队首是2号点，2号点出队。2号点可以到达4号点，我们松弛2, 4，但是4号点**已经在队列里**了，所以4号点就不入队了（之后解释原因）。




```
Q : <3, 4>
dist[4] = 3
```

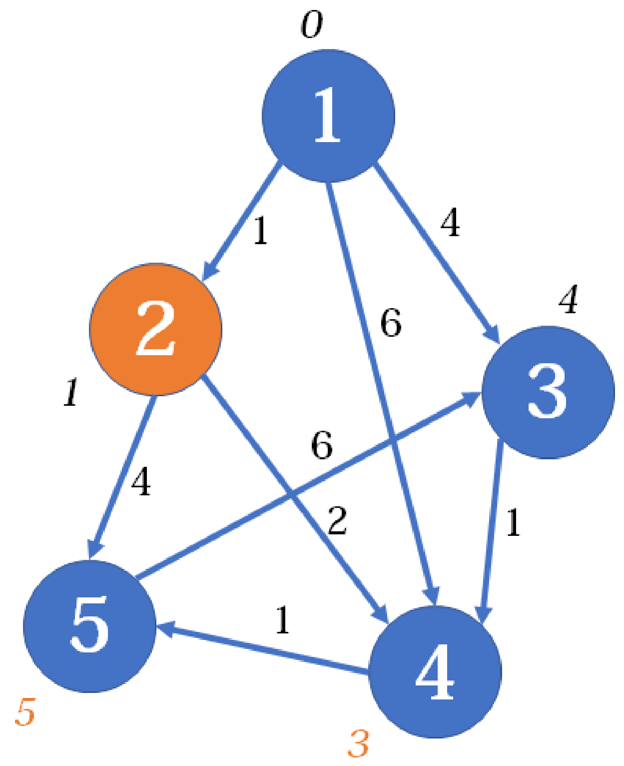
因为这张图非常简单，后面的流程我就不画了，无非是3号点出队，松弛3, 4，然后4号点出队而已。当队列为空时，流程结束。

为了表明SPFA的优越性，我们再来看一个稍微复杂一点的图（在原图基础上增加一个5号点）：



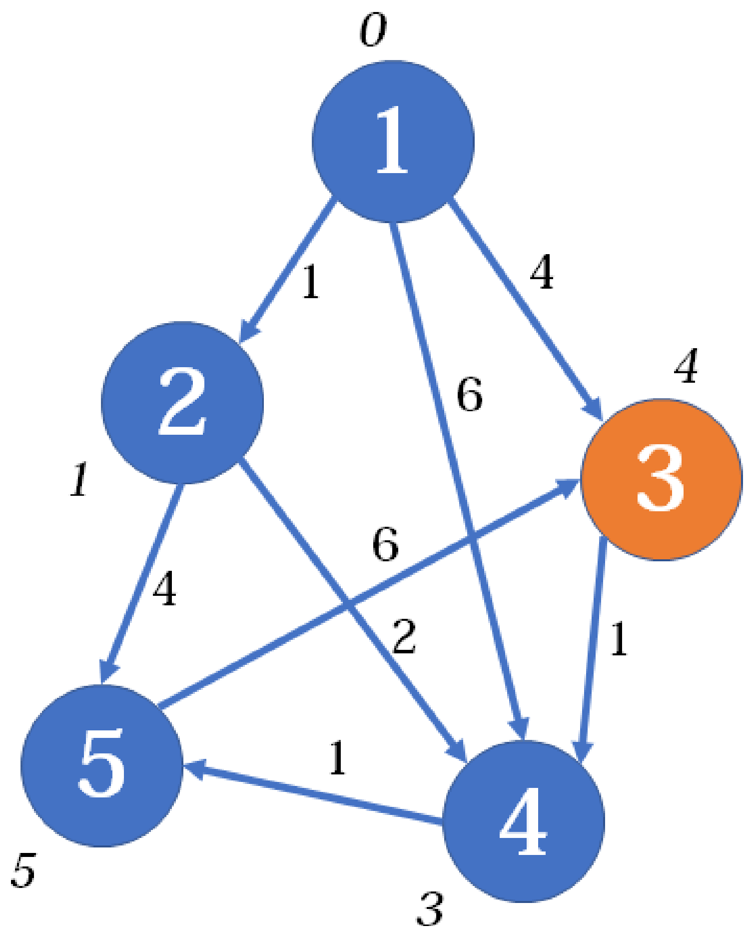
这张图，按照Bellman-Ford算法，需要松弛8*4=32次。现在我们改用SPFA解决这个问题。

显然前几步跟上次是一致的，我们松弛了1, 2、1, 3、1, 4，现在队首元素是2。我们让2出队，并松弛2, 4、2, 5。5未在队列中，5入队。



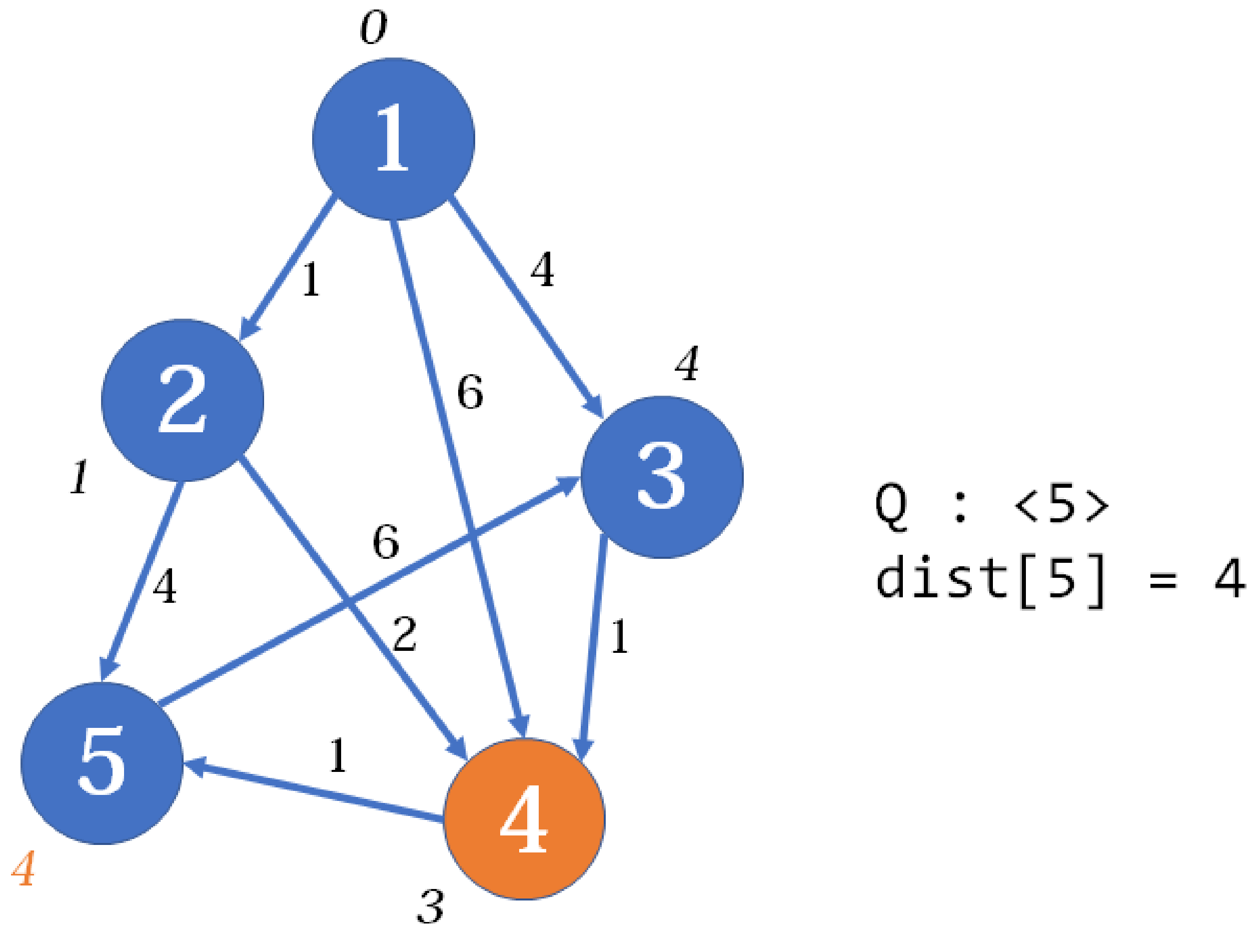

```
Q : <3, 4, 5>
dist[4] = 3
dist[5] = 5
```

3号点没能更新什么东西：




```
Q : <4, 5>
```

然后4号点出队，松弛4, 5，然后5号点已在队列所以不入队。



最后5号点出队，dist[3]**未被更新**，所以3号点通往的点**不会跟着被更新**，因此3号点不入队，循环结束。

这个过程中，我们只进行了**6次松弛**，远小于B-F算法的32次，虽然进行了入队和出队，但在n、m很大时，SPFA通常还是显著快于B-F算法的。·（据说随机数据下期望时间复杂度是 $O(m+n\log n)$ ）

总结一下，SPFA是如何做到“只更新可能更新的点”的？

只让当前点能到达的点入队
如果一个点**已经在队列**里，便**不重复入队**
如果一条边**未被更新**，那么它的终点不入队

原理是，我们的目标是松弛完 $S\to P_1\to P_2\to\cdots D$ ，所以我们先把 S 能到达的所有点加入队列，则 P_1 一定在队列中。然后对于队列中每个点，我们都把它能到达的所有点加入队列（不重复入队），这时我们又可以保证 P_2 一定在队列中。另外注意到，假如 $P_i\to P_{i+1}$ 是目标最短路上的一段，那么在松弛这条边时它一定是会被更新的，所以如果一条边未被更新，它的终点就不入队。

我们用一个inqueue[]数组来记录一个点是否在队列里，于是SPFA的代码如下：

```
void SPFA(int s)
{
    queue<int> Q;
    Q.push(s);
    while (!Q.empty())
    {
        int p = Q.front();
        Q.pop();
        inqueue[p] = 0;
        for (int e = head[p]; e != 0; e = edges[e].next)
        {
            int to = edges[e].to;
            if (dist[to] > dist[p] + edges[e].w)
            {
                dist[to] = dist[p] + edges[e].w;
                if (!inqueue[to])
                {
                    inqueue[to] = 1;
                    Q.push(to);
                }
            }
        }
    }
}
```

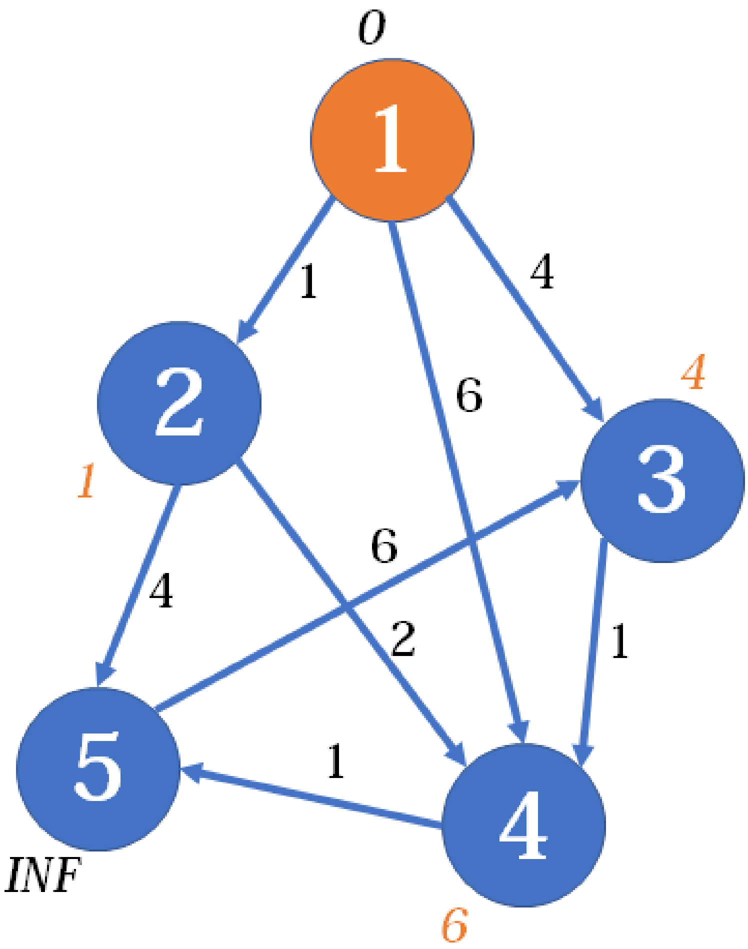
这个算法已经可以A掉洛谷P3371的单源最短路径（弱化版）了。然而它的时间复杂度**不稳定**，最坏情况可以被卡成Bellman-Ford，也就是 $O(mn)$ 。现在不少最短路的题会刻意卡SPFA，所以会有大佬说：SPFA死了。然而这仍然不失为一种比较好写、通常也比较快的算法。

SPFA也可以判负权环，我们可以用一个数组记录每个顶点进队的次数，当一个顶点**进队超过n次**时，就说明存在负权环。（这与Bellman-Ford判负权环的原理类似）

Dijkstra算法

下面介绍一种复杂度稳定的算法：Dijkstra算法。

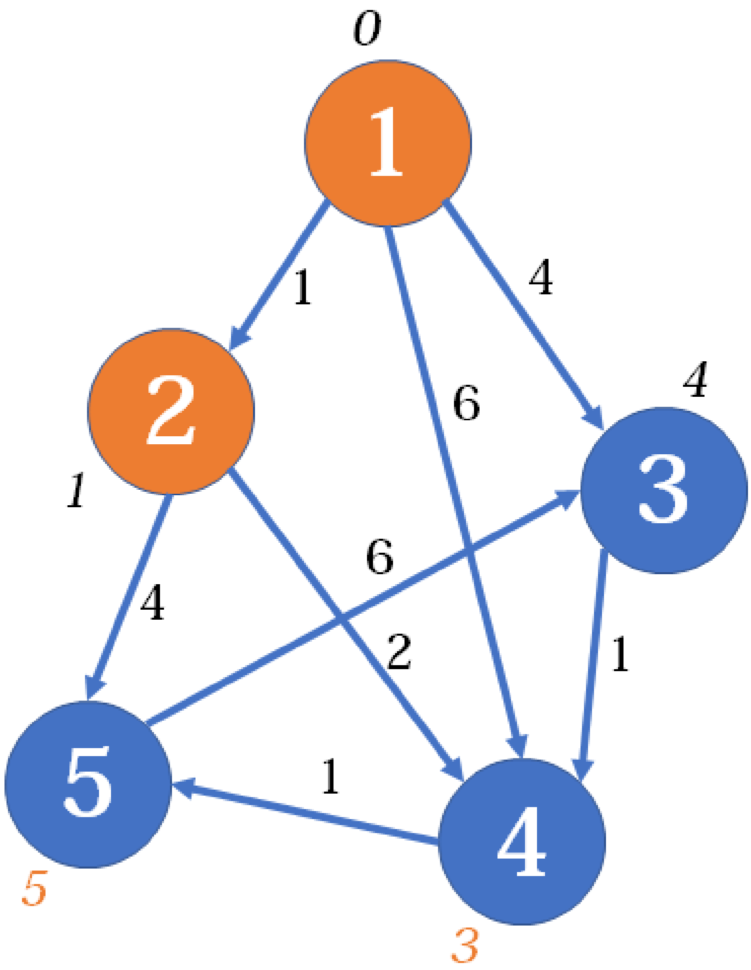
Dij基于一种**贪心**的思想，我们假定有一张没有**负边**的图。首先，起点到起点的距离为0，这是没有疑问的。现在我们对起点和它能直接到达的所有点进行松弛。




```
dist[2] = 1
dist[3] = 4
dist[4] = 6
```

因为没有负边，这时我们可以肯定，**离起点最近的那个顶点的dist一定已经是最终结果**。为什么？因为没有负边，所以不可能经由其他点，使起点到该点的距离变得更短。

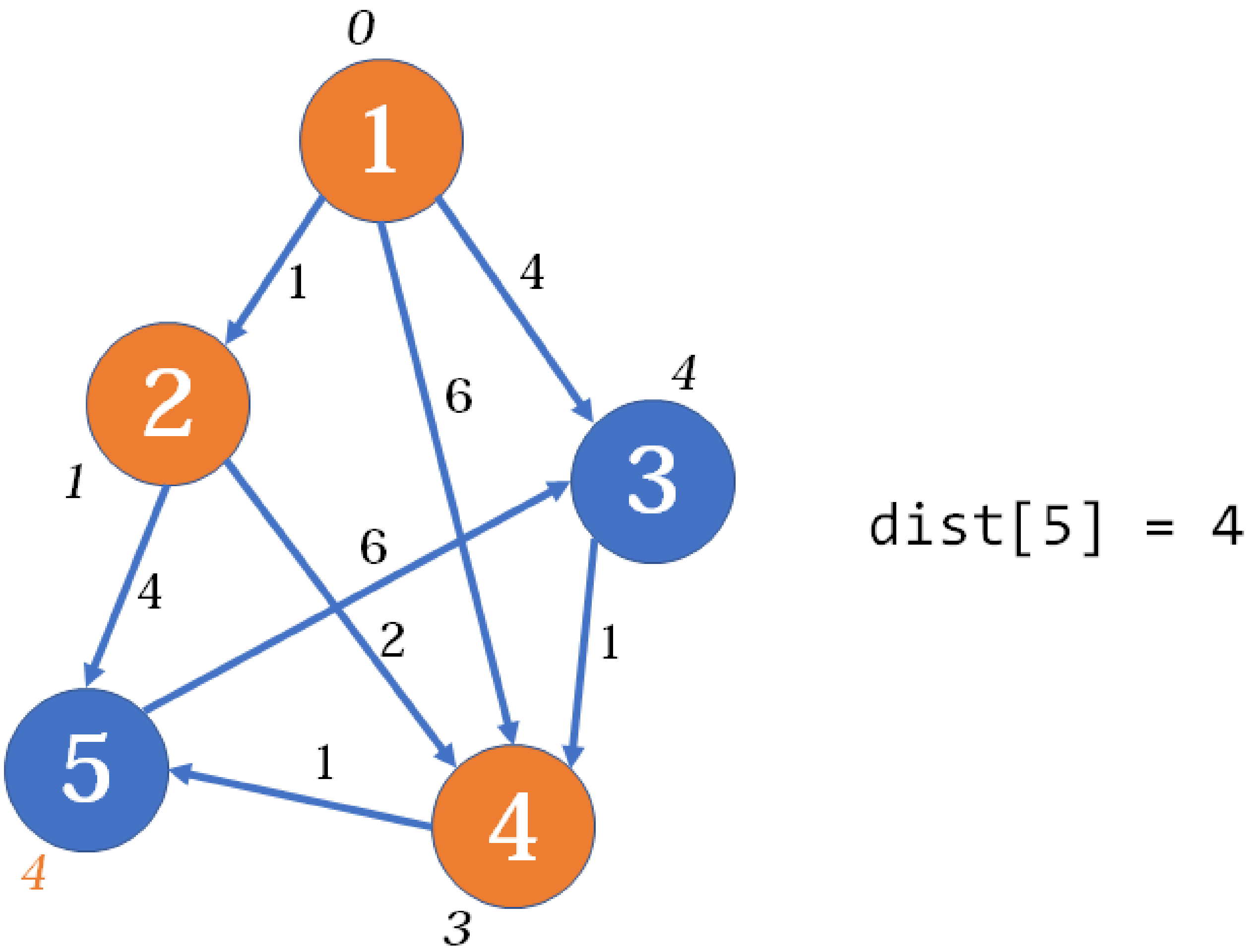
那现在我们来考察2号点：




```
dist[4] = 3
dist[5] = 5
```

我们对2号点和它能到达的点进行松弛。这时dist保存的是起点**直接到达**或**经由2号点到达**每个点的最短距离。我们这时候取出未访问过的dist最小的点（即4号点），这个点的dist也不可能变得更短了（因为其他路径都至少要从起点直接到达、或者经由2号点到达另一个点，再从这另一个点到达4号点）。

继续这个流程，松弛4号点能到达的点：



然后分别考察3、5号点，直到所有点都被访问过即可。

总结一下，Dijkstra算法的流程就是，不断取出**离顶点最近**而**没有被访问过**的点，松弛它和它能到达的所有点。

如何取出离顶点最近的点？如果暴力寻找，那就是朴素的Dijkstra算法，时间复杂度是 $O(n^2)$ ，但我们可以采取**堆优化**。具体而言，我们可以用一个**优先队列**（或手写堆，那样更快）来维护所有节点。这样可以实现在 $O(m\log m)$ 的时间内跑完最短路[1]。

首先写一个结构体：

```
struct Polar{
    int dist, id;
    Polar(int dist, int id) : dist(dist), id(id){}
};
```

然后写一个仿函数（也可以用重载Polar的小于运算符代替），再构建优先队列：

```
struct cmp{
    bool operator ()(Polar a, Polar b){ // 重载()运算符，使其成为一个仿函数
```

```
        return a.dist > b.dist;    // 这里是大于，使得距离短的先出队
    }
};
priority_queue<Polar, vector<Polar>, cmp> Q;
```

Dijkstra算法的实现：

```
void Dij(int s)
{
    dist[s] = 0;
    Q.push(Polar(0, s));
    while (!Q.empty())
    {
        int p = Q.top().id;
        Q.pop();
        if (vis[p])
            continue;
        vis[p] = 1;
        for (int e = head[p]; e != 0; e = edges[e].next)
        {
            int to = edges[e].to;
            dist[to] = min(dist[to], dist[p] + edges[e].w);
            if (!vis[to])
                Q.push(Polar(dist[to], to));
        }
    }
}
```

很多人可能像我一开始一样，会试图这么写：

```
//错误代码
struct cmp
{
    bool operator()(int a, int b)
    {
        return dist[a] > dist[b];
    }
};
priority_queue<int, vector<int>, cmp> Q;
```

这样看起来是省了写结构体的工夫，然而**这是错误的**，因为这种写法**破坏了堆的结构**，A进优先队列时比B小，可能出队时就比B大了。一定要注意，**堆中元素的大小关系必须保持不变。**

当然，也有一种简化的写法，利用STL里的pair：

```
typedef pair<int, int> Pair;
priority_queue<Pair, vector<Pair>, greater<Pair> > Q;
```

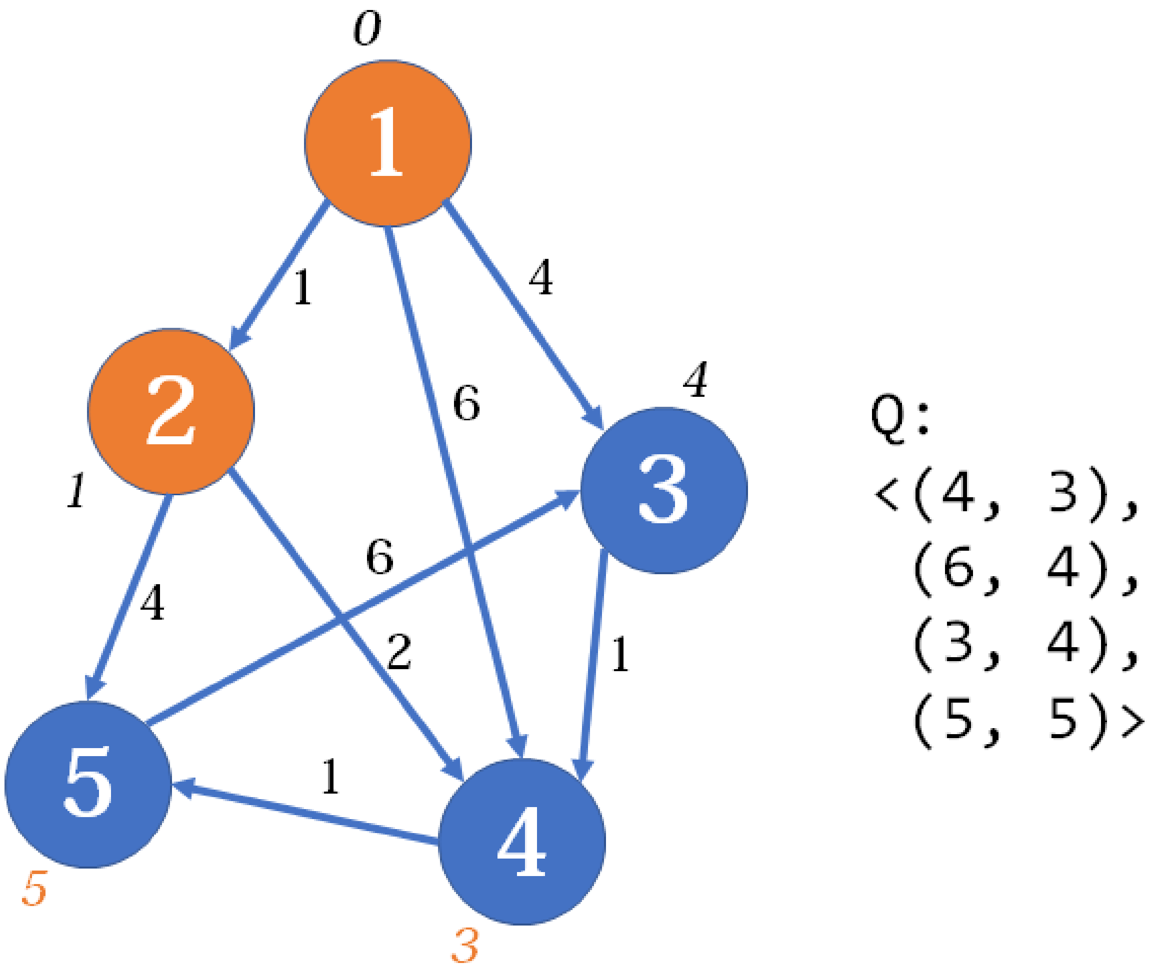
这样的代码与原来只有三行的区别：

```
void Dij(int s)
{
    dist[s] = 0;
    Q.push(make_pair(0, s));
    while (!Q.empty())
    {
        int p = Q.top().second;
        Q.pop();
        if (vis[p])
            continue;
        vis[p] = 1;
        for (int e = head[p]; e != 0; e = edges[e].next)
        {
            int to = edges[e].to;
            dist[to] = min(dist[to], dist[p] + edges[e].w);
            if (!vis[to])
                Q.push(make_pair(dist[to], to));
        }
    }
}
```

但还是省去了写结构体和仿函数的步骤，因为pair已经内建了比较函数。

也许你会想，每个步骤不是应该取**当前**离源点最近、且未被访问过的元素吗，但我们现在每次让一个pair进入优先队列，这时pair里面存储的dist是**那时**该点到源点的距离，我怎么能保证每次取出来的点恰是离源点最近的点呢？

其实是这样的，在一个点被访问前，优先队列里会存储这个点被更新的整个**历史**。比如下面这个状态，队列里既有(6, 4)又有(3, 4)，但是(3, 4)会比(6, 4)先出队，等到(6, 4)出队的时候，4这个点已经被访问了，所以不会有影响。



注意：堆优化Dij虽然复杂度稳定且较低，但是不能处理**负边**。原因很明显，如果有负边，就不能保证离源点最近的那个点的dist不会被更新了。

打印路径

我们之前只是求出了最短路径长，如果我们要打印具体路径呢？这听起来是一个比较困难的任务，但其实很简单，我们只需要用一个pre[]数组存储每个点的**父节点**即可。（单源最短路的起点是固定的，所以每条路有且仅有一个祖先节点，一步步溯源上去的路径是唯一的。相反，这里不能存**子节点**，因为从源点下去，有很多条最短路径）

每当更新一个点的dist时，顺便更新一下它的pre。这种方法对SPFA和Dij都适用，以下是对SPFA的修改：

```
if (edges[e].w + dist[p] < dist[to])
{
    pre[to] = p;     //在这里加一行
    dist[to] = edges[e].w + dist[p];
    if (!inqueue[to])
    {
        Q.push(to);
```

```
        inqueue[to] = 1;
    }
}
```

打印（以打印从1到4的最短路为例）：

```
int a = 4;
while (a != 1)
{
    printf("%d<-", a);
    a = pre[a];
}
printf("%d", a);
```

这样打印出的结果是反向的箭头，如果想得到正向的箭头，可以先将结果压入数组再逆序打印。