最短路

概念解读

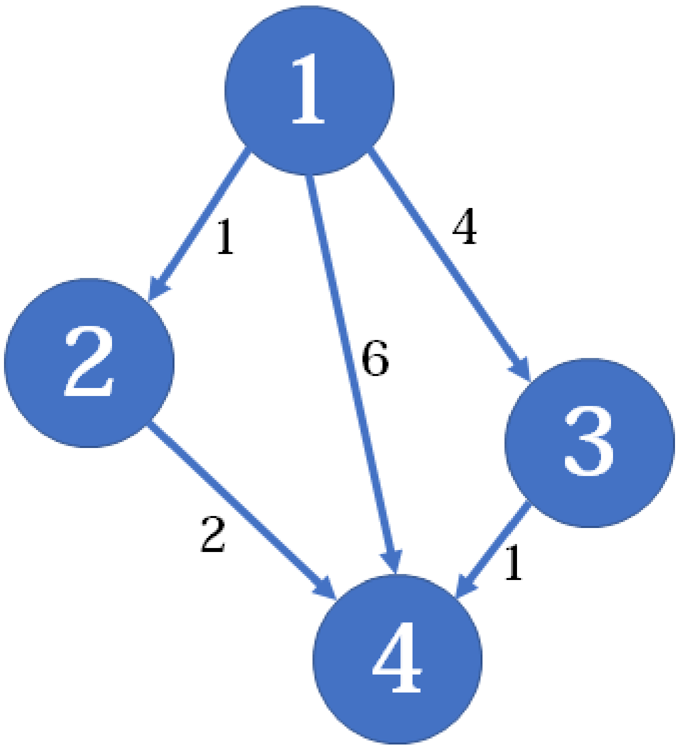
- 路径
- 最短路
- 有向图中的最短路、无向图中的最短路
- 单源最短路、每对结点之间的最短路

基本性质

对于边权为正的图，任意两个结点之间的最短路，不会经过重复的结点。

对于边权为正的图，任意两个结点之间的最短路，不会经过重复的边。

对于边权为正的图，任意两个结点之间的最短路，任意一条的结点数不会超过 n，边数不会超过 $n-1$
。



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

我们一般用Floyd算法解决**多源最短路**问题。

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）。

```
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]);
}
```

Floyd的**枚举顺序**一般都是大家头痛的点，我们将在下面介绍如何证明最外围应该是枚举顺序：

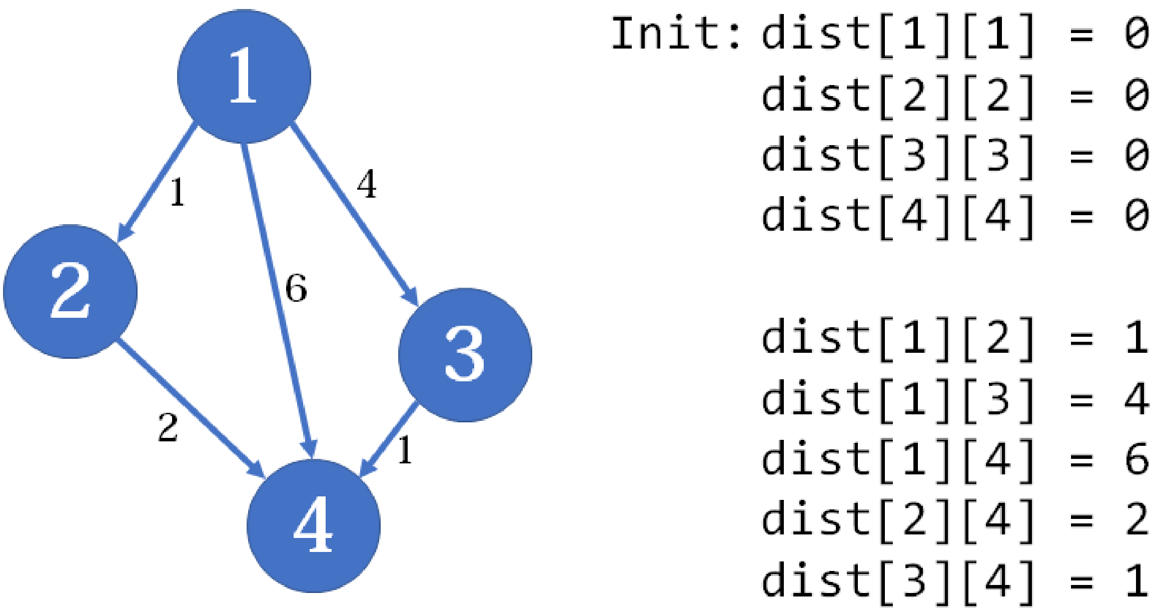
- 假设有两个点 a_1, a_n ，它们之间的最短路我们可以用**点集** $a_1, a_2, \cdots a_n$ 来表示
- 设其中**编号最小**的点为 a_m ，我们首先会在 $k = m$ 时枚举到

  $dis[a_{m-1}][a_{m+1}] = \min(dis[a_{m-1}][a_{m+1}], dis[a_{m-1}][a_m] + dia[a_m][a_{m+1}])$

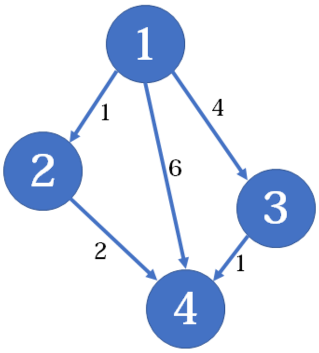
  显然，这就是点 a_{m-1} 到点 a_{m+1} 之间的**最短距离**，如果不是的话，就和我们之前最短路的假设矛盾（可以用这个**更短路径**替换原先的 a_{m-1}, a_m, a_{m+1} 这一段）
- 此时，我们可以将 a_{m-1}, a_{m+1} 之间的边长**视为** $dis[a_{m-1}][a_{m+1}]$，相当于我们在原序列中把 a_m 这个点丢弃了
- 重复上述过程，直到剩下起点和终点为止

接下来我们以上图为例，看看Floyd算法的具体过程：

初始化：

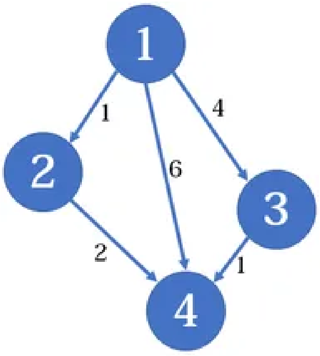


第一趟，$k = 1$：



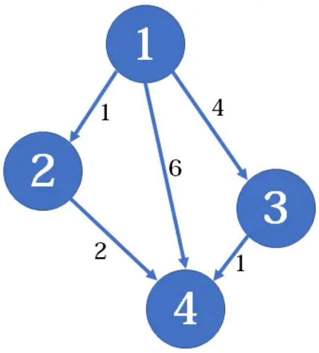

```
k=1
dist[1][1] = min(dist[1][1], dist[1][1]+dist[1][1])
dist[1][2] = min(dist[1][2], dist[1][1]+dist[1][2])
dist[1][3] = min(dist[1][3], dist[1][1]+dist[1][3])
dist[1][4] = min(dist[1][4], dist[1][1]+dist[1][4])
dist[2][1] = min(dist[2][1], dist[2][1]+dist[1][1])
...
```

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])
...
```

第三趟，$k=3$:



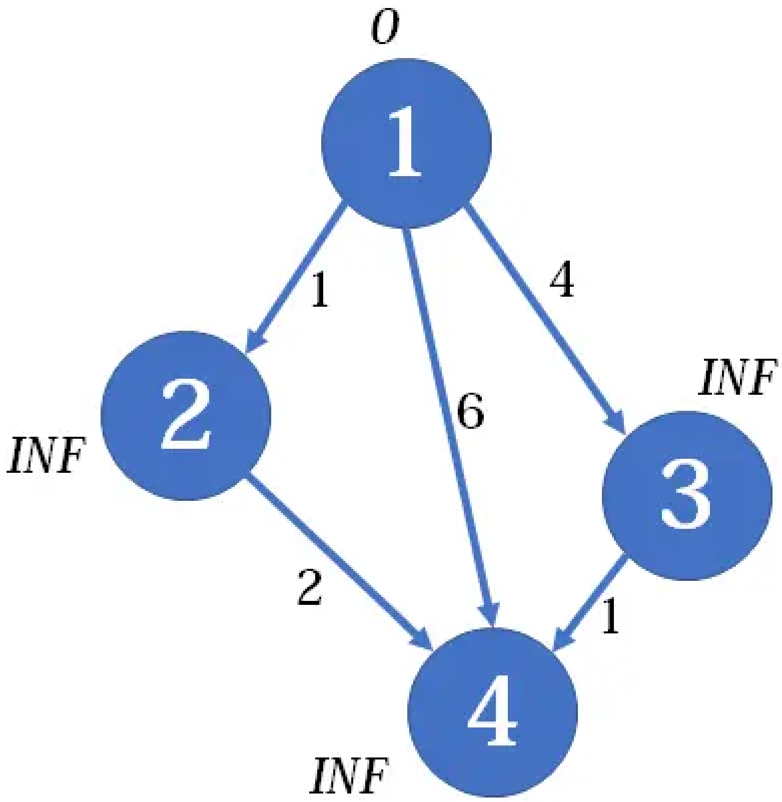

```
k=3
dist[1][1] = min(dist[1][1], dist[1][3]+dist[3][1])
dist[1][2] = min(dist[1][2], dist[1][3]+dist[3][2])
dist[1][3] = min(dist[1][3], dist[1][3]+dist[3][3])
dist[1][4] = min(dist[1][4], dist[1][3]+dist[3][4])
dist[2][1] = min(dist[2][1], dist[2][3]+dist[3][1])
...
```

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，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 $S, p_1, p_2, \cdots p_m, D$ 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 。

松弛

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，这个**松弛操作**是对于边 (u, v) 进行的

```
// 此处e[x][y] 表示 xy之间边的距离
dist[y] = min(dist[x] + e[x][y])
```

显然，对于我们起点为 S ，终点为 D 的最短路 $S, p_1, p_2, \cdots p_m, D$ 而言，我们首先需要松弛 (S, p_1) ，接着需要松弛 (p_1, p_2) ，以此类推

此时有同学可能就有疑问了，我们怎么知道 $p_1, p_2, \cdots p_m$ 呢?

关键的来了，Bellman-Ford算法告诉我们：**把所有边松弛一遍!**

由于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, p_1 已经被松弛过了，现在我们要松弛 p_1, p_2 ，怎么做呢?

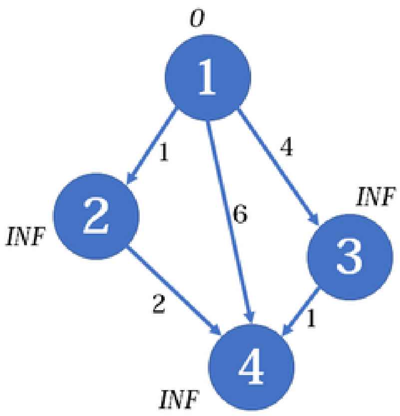
再把所有边松弛一遍!

以此类推，在前面，我们已经证明了最短路上的点的总个数一定**不大于** n ，因此，我们至多进行 $n-1$ 次松弛，就可以得到所有点到起点的最短距离了

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 $O(nm)$

```cpp
void Bellman_Ford(int n, int m) {
    for (int j = 1; j < n; j++) for (int u = 1; u <= n; u++)
    for (auto [v, w]: e[u]) {
        // c++17 above
        cmin(dist[v], dist[u] + w);
    }
}
```

上文使用的是**邻接表存图**，我们在下图中更形象地看一下：



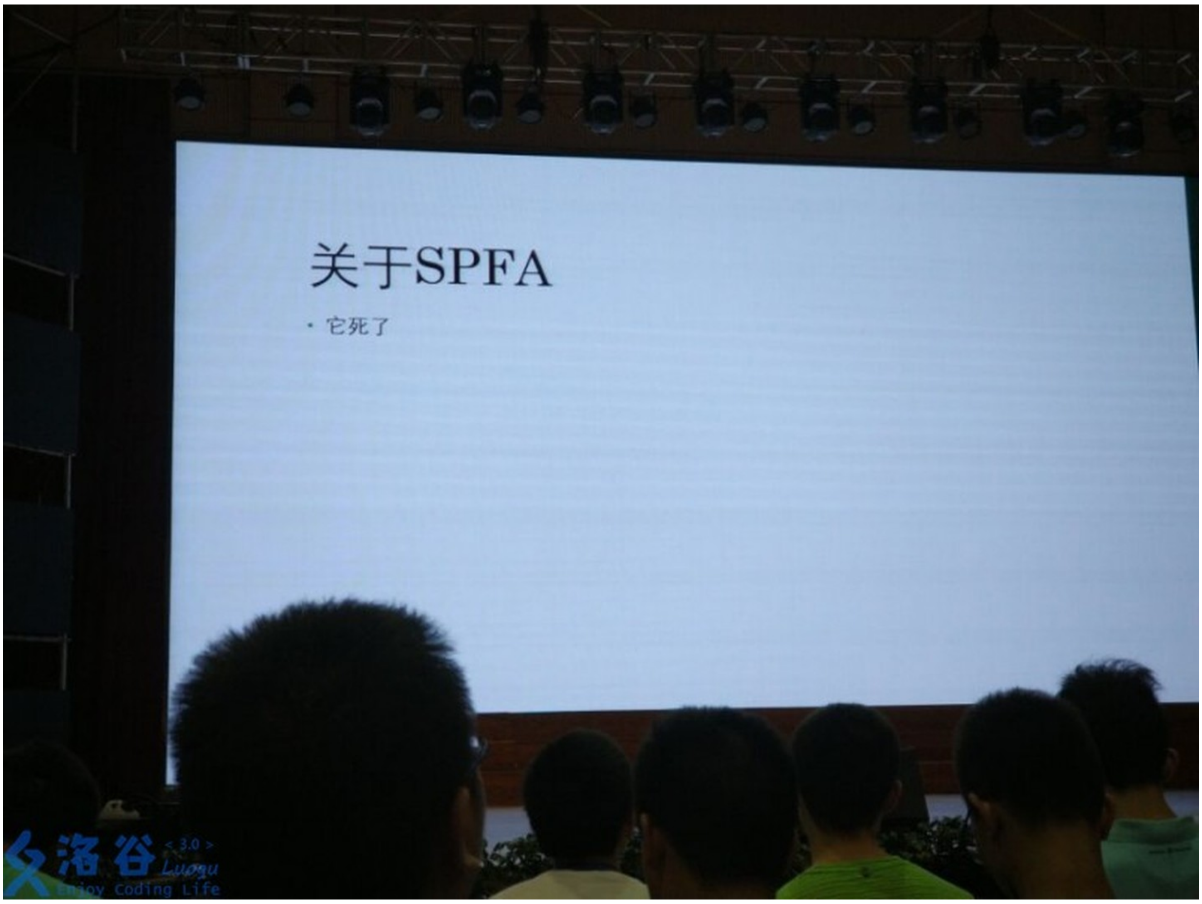

```
1->2  dist[2] = min(INF, 0+1) = 1
1->3  dist[3] = min(INF, 0+4) = 4
1->4  dist[4] = min(INF, 0+6) = 6
2->4  dist[4] = min(6, 1+2) = 3
3->4  dist[4] = min(3, 4+1) = 3
```

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，只需要在进行完成 $n-1$ 次松弛后，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（Shortest Path Fast Algorithm)

在看SPFA算法前，我们先看一张图：



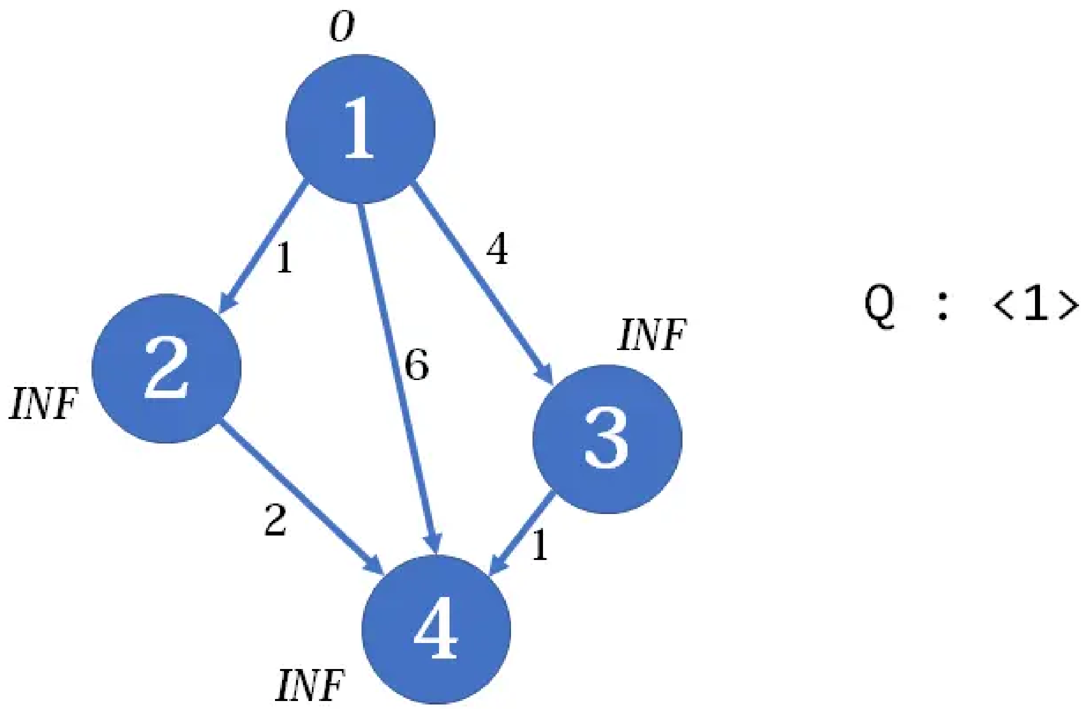
对于**正权图**而言，SPFA确实死了，但是世界上的图不只有正权图，因此，我们还是需要学习SPFA

在Bellman-Ford算法中，$O(nm)$ 的时间复杂度，只要来源于我们**每次**松弛的过程中，都可能进行了很多无意义操作

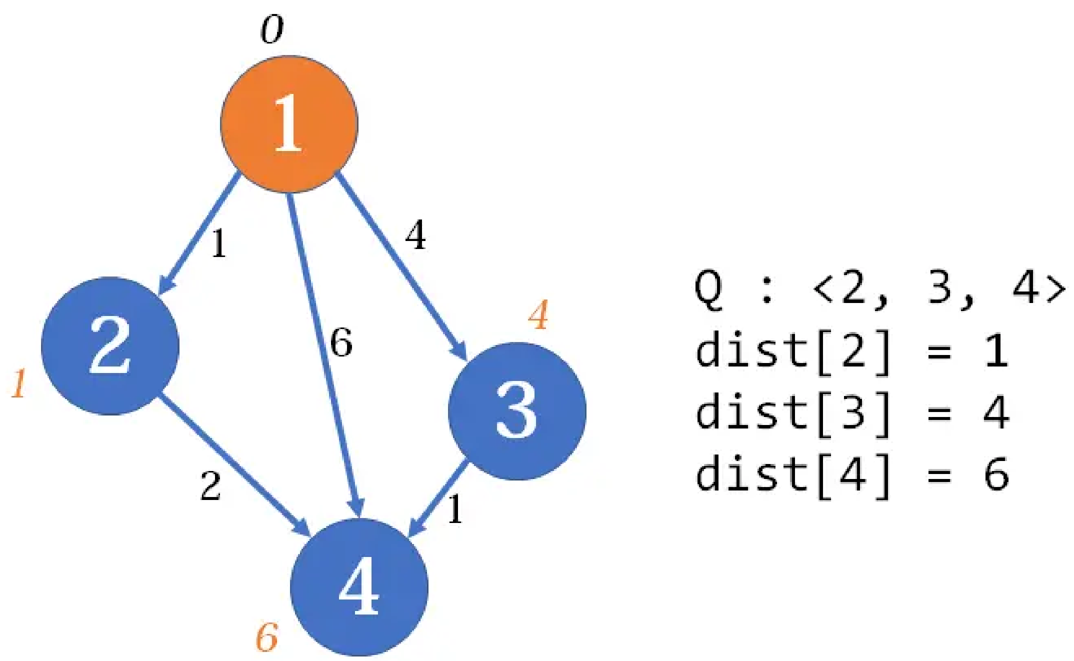
- 例如某一次松弛操作后，某个点 u 的距离 $dist[u]$ 没有发生改变，那么我们下一轮再对**以它**为起点的边进行松弛就没有意义

SPFA就是利用了这种思想，规避了许多无意义的松弛操作，算法的具体流程如下：

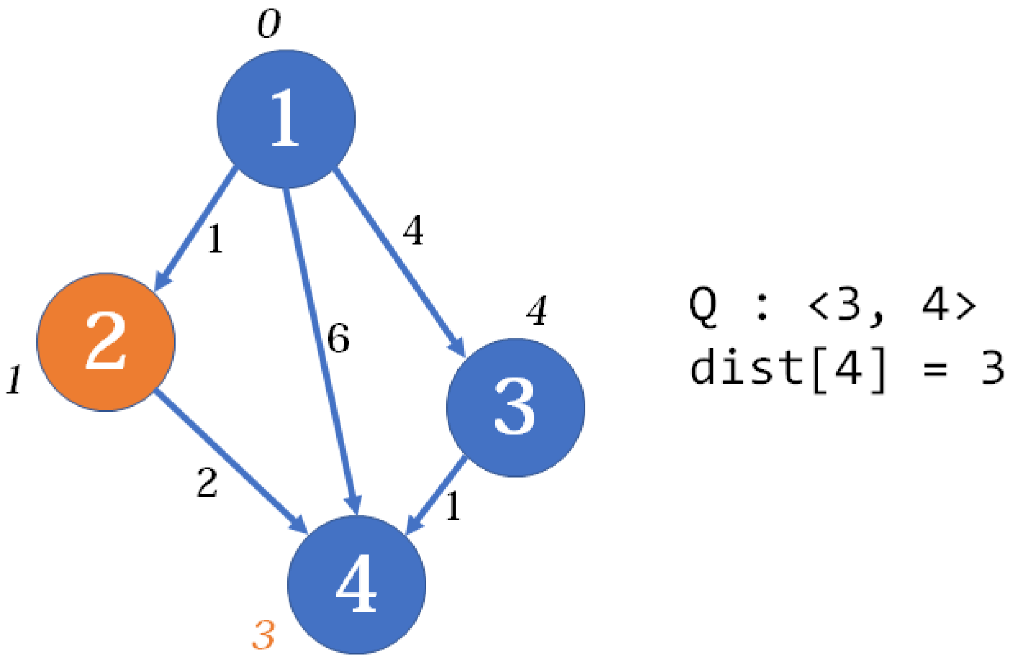
- 维护一个队列，一开始，把起点放进队列



- 现在我们对 1 号点伸出的所有边进行松弛，发现 $2,3,4$ 号点`dist`被更新，因此将它们加入队列

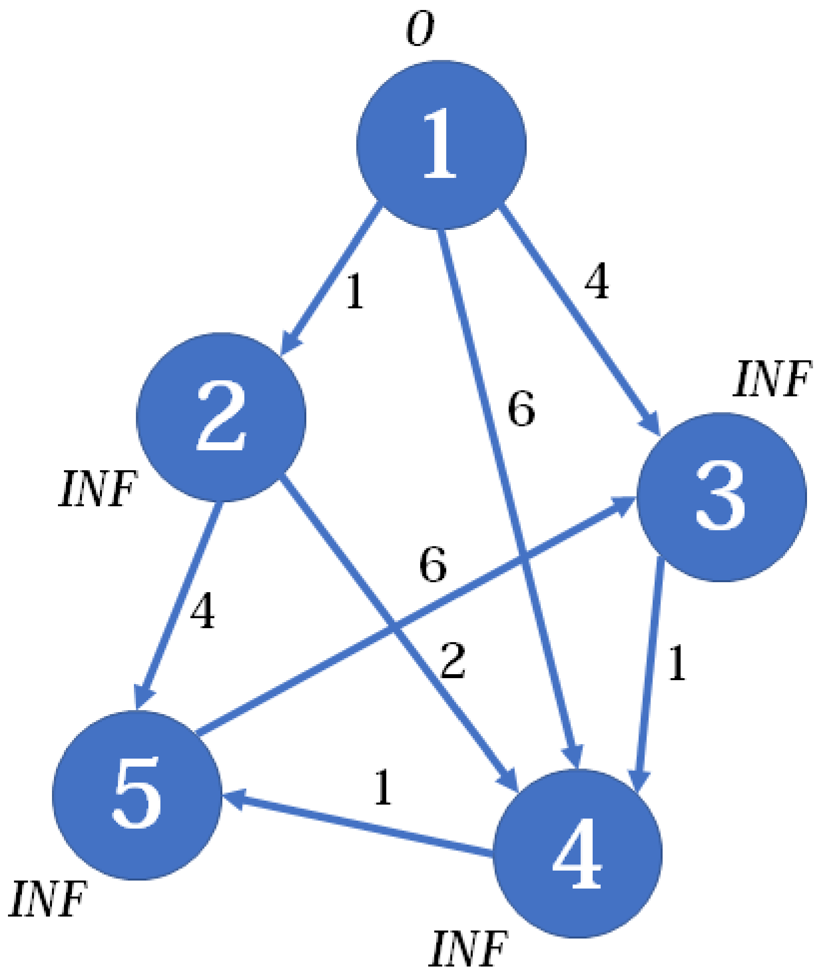


- 现在队首是 2 号点，2 号点出队。2 号点可以到达 4 号点，我们松弛 $(2,4)$，但是 4 号点**已经在队列里**了，所以 4 号点就不入队了



- 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。

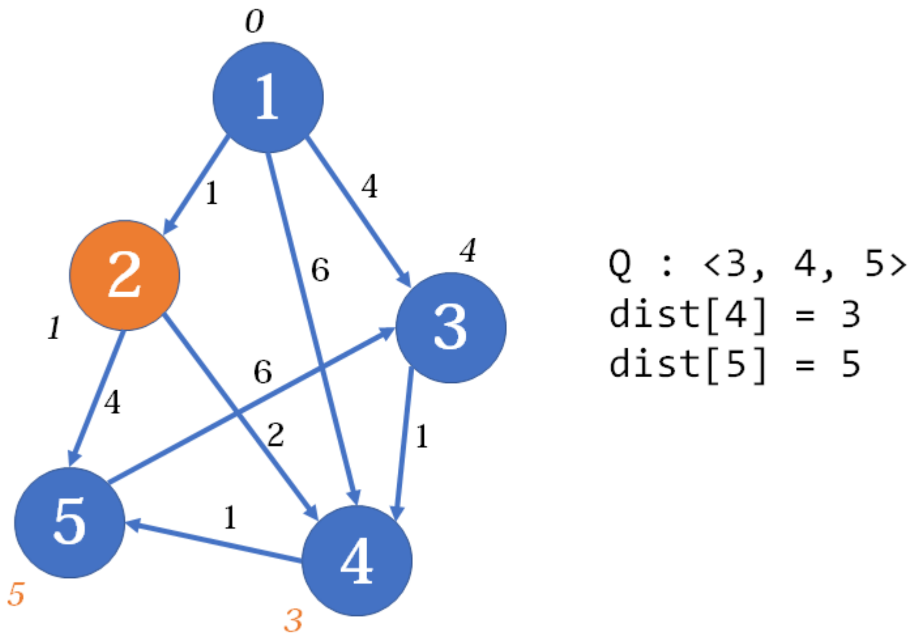
为了表示SPFA的优越性，我们再以下图为例：



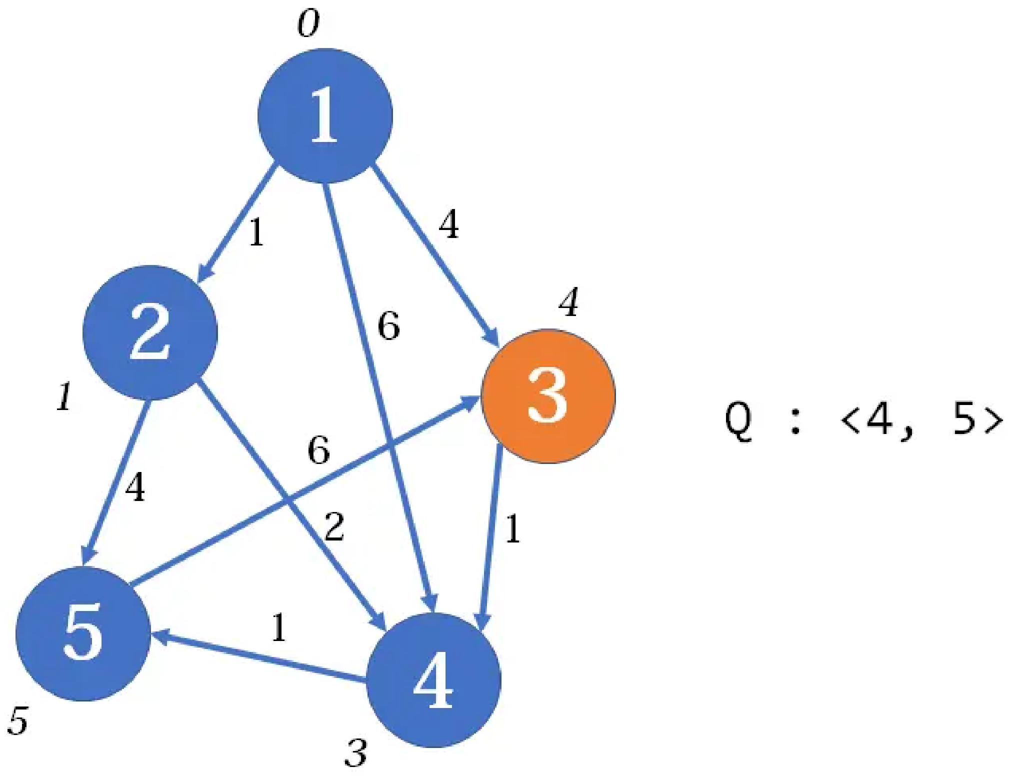
这张图，按照Bellman-Ford算法，需要松弛 $4 \times 8 = 32$ 次。现在我们改用SPFA解决这个问题

- 显然，前两步操作是和上图一样，这里就不再赘述

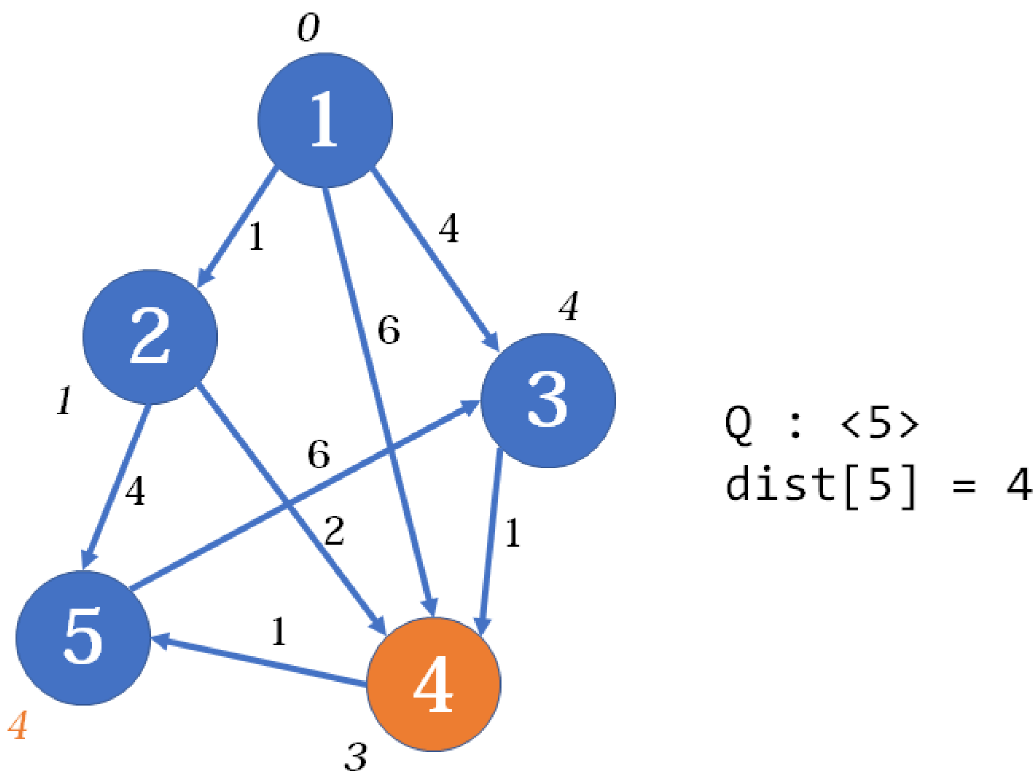
- 现在队首元素是 2 。我们让2出队，并松弛 $(2,4)$、$(2,5)$。5 未在队列中，5 入队



- 3 号点没能更新什么东西



- 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 很大的情况下，SPFA通常还是显著快于 B-F 算法的

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 $S \to p_1 \to p_2 \cdots \to p_m \to D$，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。

对于**不重复入队**操作，我们可以用一个 `inq[]` 数组进行存储

以上，我们得到了SPFA的代码：

```
// vector<pair<int, int>> e
// s - 源点
void spfa (int s) {
    dis[s] = 0;
    queue<int> q; q.push(s);
```

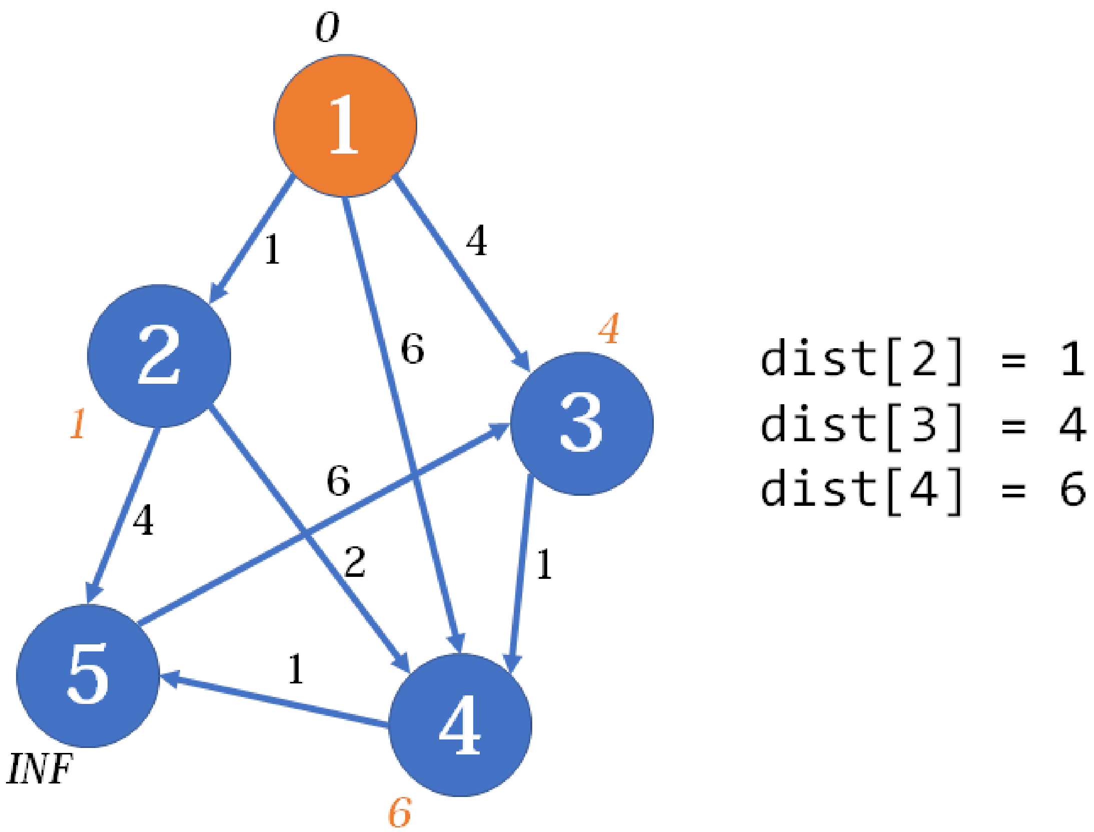
```
    while(!q.empty()) {
        auto u = q.front();
        q.pop(); inq[u] = 0;
        for(auto [v, w]: e[u]) if (dis[u] + w < dis[v]) {
            dis[v] = dis[u] + w;
            if (!inq[v]) inq[v] = 1, q.push(v);
        }
    }
}
```

SPFA也可以判负权环，我们可以用一个 `cnt` 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

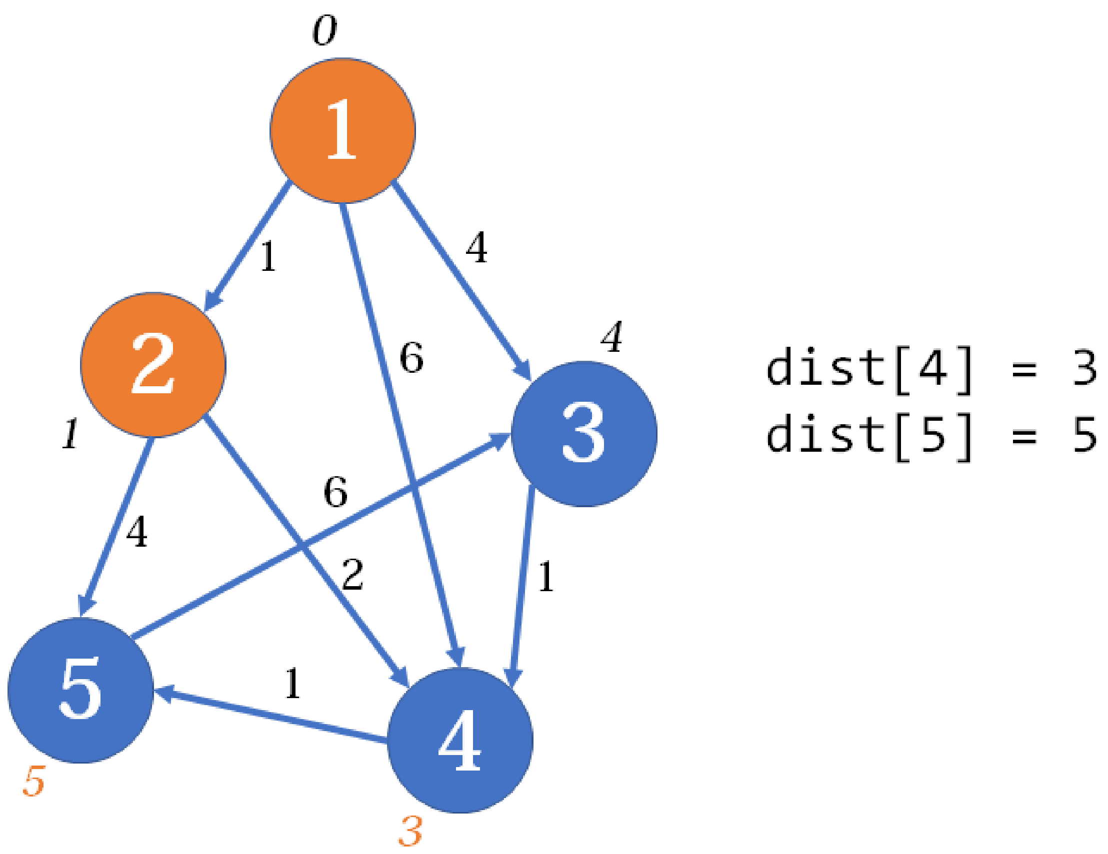
下面介绍一种复杂度稳定的算法：Dijkstra算法。

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。



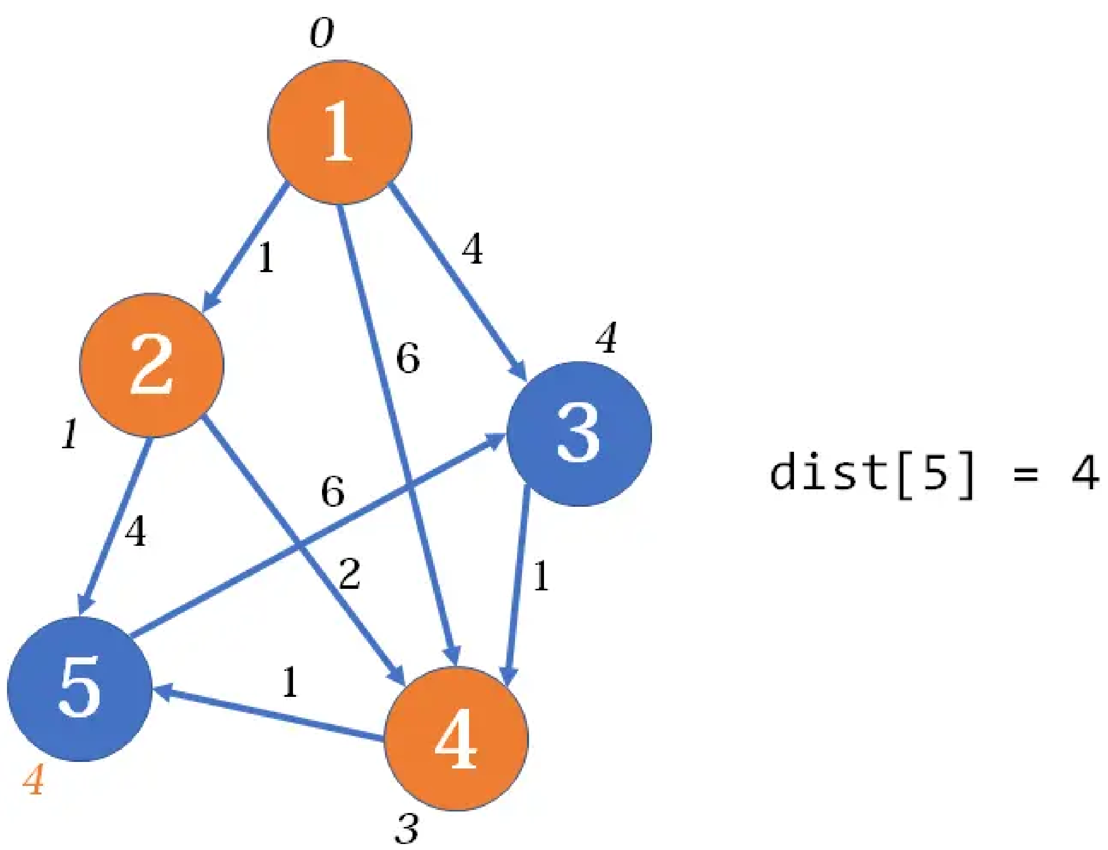
因为没有负边，这时我们可以肯定，**离起点最近的那个顶点的** `dist` **一定已经是最终结果**。为什么？因为没有负边，所以不可能经由其他点，使起点到该点的距离变得更短。

那现在我们来考察 2 号点：



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 $O(mlogm)$ 的时间内跑完最短路

由于维护节点时，我们需要以**距离**为关键字进行排序，因此，在使用 `std::pair` 时，需要将距离放在 `first` ，编号放在 `second`

```
void dij(int s) {
    fill(all(dis), INF); dis[s] = 0;
    fill(all(vis), 0);
    priority_queue<pair<int, int>, vector<pair<int, int>>,
                   greater<pair<int, int>>> pq;
    pq.emplace(0, s);

    while(!pq.empty()) {
        // c++17
        auto [_, u] = pq.top(); pq.pop();
        if (vis[u]) continue; vis[u] = 1;
        // c++17
        for (auto [v, w]: e[u]) {
            if (dis[u] + w < dis[v]) {
                dis[v] = dis[u] + w;
                pq.emplace(dis[v], v);
            }
        }
    }
};
```

值得注意的是，在边的数量接近 $O(n^2)$ ，即为稠密图时，**优先队列优化**的Dijkstra算法，会比朴素的Dijkstra算法**更慢**

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

```
// 维护
if (dis[u] + w < dis[v]) {
    dis[v] = dis[u] + w;
    pre[v] = u;
    pq.emplace(dis[v], v);
}
```

```
// 打印
// 对于打印起点到终点的路径，我们可以用vector存储，倒序输出即可
while (t != s) {
    cout << t << ' ';
    t = pre[t];
}
cout << s << endl;
```

差分约束

差分约束系统是下面这种形式的多元一次不等式组（$y_1, y_2 \cdots$ 为已知量）：

$$\begin{cases} x_{v1} - x_{u1} \le y_1 \\ x_{v2} - x_{u2} \le y_2 \\ \cdots \\ x_{vn} - x_{un} \le y_n \end{cases}$$

每个不等式称为一个**约束条件**，都是两个未知量之差小于或等于某个**常数**

在算法竞赛中，很多题目会给出（或隐性地给出）一系列的**不等关系**，我们可以尝试把它们转化为差分约束系统来解决。

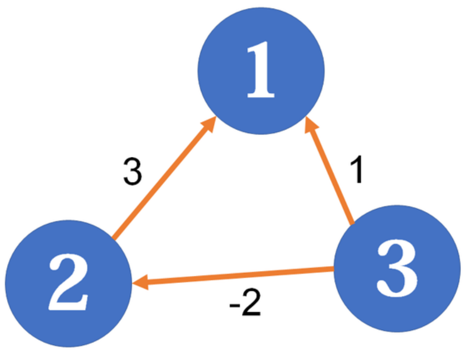
我们接下来以一个简单的模型为例进行讨论：

$x_1 - x_2 \le c$，即 $x_1 \le x_2 + c$

这个不等式与**最短路问题**中的三角形不等式 $dis[v] \le dis[u] + w(u, v)$ 非常类似，利用这一点，我们可以把它转化为一个**图论**问题。也就是说，对于每一个 $x_{v_i} - x_{ui} \le y_i$，我们都可以建一条 $< u_i, v_i, y_i >$ 的**有向边**

这样建出的**有向图**，它的每个**顶点**都对应差分约束系统中的一个**未知量**，源点到每个顶点的**最短路**对应这些未知量的**值**，而每条**边**对应一个**约束条件**

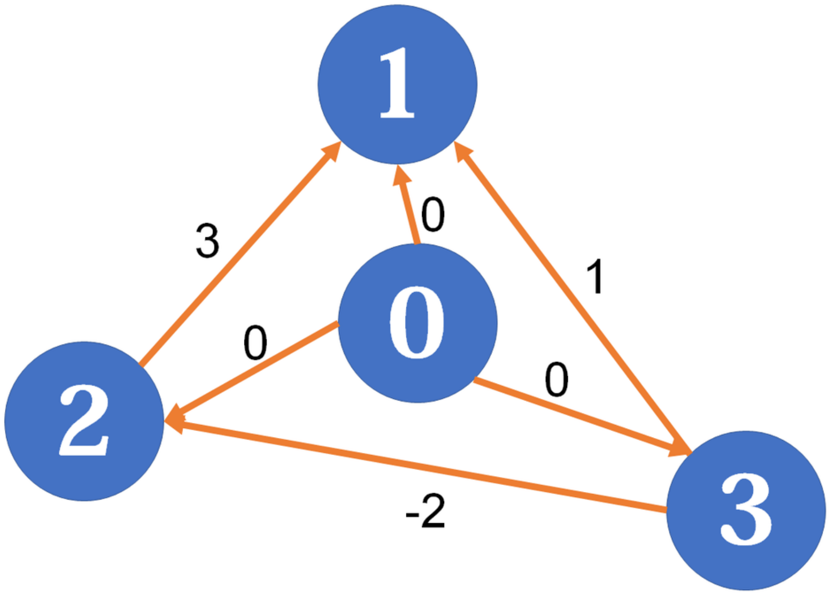
$$\begin{cases} x_1 - x_2 \le 3 \\ x_2 - x_3 \le -2 \\ x_1 - x_3 \le 1 \end{cases}$$



那么问题来了，既然是最短路，**源点**在哪里呢?

实际上取哪个点为源点是无关紧要的，但是，有时候我们随意选取源点后，得到的图不是连通的（如上图中的 1 点），这样求出来的结果很容易出现 `inf`。为了避免这种情形，我们习惯人为地增加一个**超级源点**。

例如我们现在人为地新增一个 0 号点（或 $n+1$ 号点），从它向所有顶点连一条边权为 0 的边：



现在我们以 0 号点为源点求各点的最短路即可。注意，这相当于了添加了以下约束条件：

$$\begin{cases} x_1 - x_0 \le 0 \\ x_2 - x_0 \le 0 \\ x_3 - x_0 \le 0 \end{cases}$$

由于 x_0 的**大小**对应的是 $dis[0]$，而通常情况下我们都有 $dis[0] = 0$，可知所有未知量均小于等于 0（反映在图形上是所有点的最短路均小于等于0）。

跑了最短路后，求出的只是**一组解**，通过简单的数学计算可知，在符合差分约束系统的一组解上**加上或减去同一个数**，得到的解同样符合原系统。

例如上图 $x_1 = 0, x_2 = -2, x_3 = 0$ 很显然是一组解，同样的，将它们全加上 a，得到的新解 $x_1 = a, x_2 = a - 2, x_3 = a$ 也很显然满足约束

若我们将 $dis[0]$（即我们的超级源点）设为某个值 w 而不是 0，那么我们就能够得到满足 $x_1, x_2 \cdots x_n \le w$ 的一组解（实际上也是**最大解**）

那如何求满足 $x_1, x_2 \cdots x_n \ge w$ 的**最小解**呢？

和求最大解类似，我们只需要将超级源点的初始值设为 w，然后求**最长路**即可（最长路满足不等式 $dis[v] \ge dis[u] + w(u, v)$）就代表着我们的某个约束条件 $x_v - x_u \ge y$

最短路算法选取

显然，有时候的边权并不为正，因此不能使用 `Dijkstra` 算法

通常选用 `SPFA` 算法为优

无解的判断

若存在下面一个不等式：

$$\begin{cases} x_1 - x_2 \le -1 \\ x_2 - x_1 \le -1 \end{cases}$$

显然无解

在我们建图时，相当于建了一条 $<2,1,-1>$ 和一条 $<1,2,-1>$ 的边，此时相当于出现了 **负环**，因此，在图中出现负环时，我们的约束系统无解

常用转换

我们只学习了简单的建边（$x_1 - x_2 \le y$ 类型），但实际上我们可以通过简单的恒等变换，将其他约束条件规约成我们的简单类型

$$\begin{aligned} x_1 - x_2 \ge y \qquad & \qquad x_2 - x_1 \le y \\ x_1 - x_2 = y \qquad & \qquad x_2 - x_1 \le y \ \wedge \ x_1 - x_2 \le y \\ x_1 - x_2 < y \qquad & \qquad x_1 - x_2 \le y - 1 \\ x_1 - x_2 > y \qquad & \qquad x_2 - x_1 \le -y - 1 \end{aligned}$$