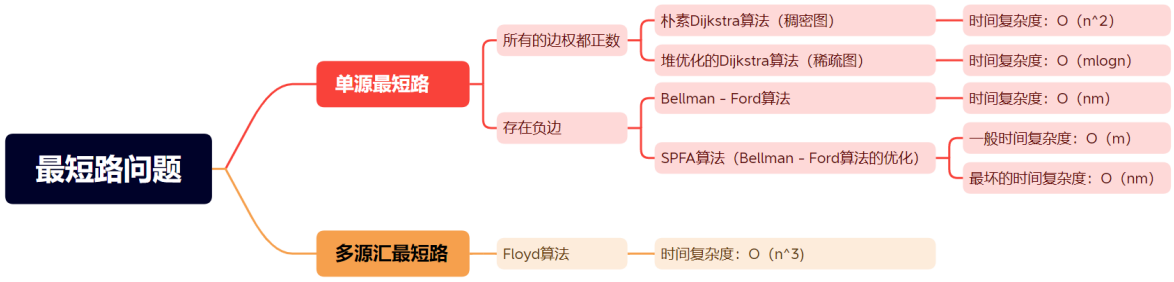
算法基础（十一）： 图 - 最短路问题1 - Dijkstra

`Dijkstra`属于图中最短路算法的一种，在图里面的最短路算法一共有`4`中，其应用的场景如下图：



几个基本概念：

1. 稠密图：一般而言，边数是顶点数的两倍即：`m ~ n ^ 2`
2. 稀疏图：边数跟顶点数是一个级别：`n ~ m`
3. 自环：从自己出发又回到自己的边
4. 重边：两个点之间有多条边，但是可能权重不一样，在最短路的问题中我们保留最短的那条边即可

注意事项：

1. 每种算法适用的场景是不同的，堆优化的`Dijkstra`算法在稠密图中，由于`m ~ n ^ 2`所以此时的时间复杂度还不如朴素的`Dijkstra`算法
2. `SPFA`算法虽然是`Belman - ford`算法的优化，但是如对最短路经过的边数进行限制，比如要求经过的边数小于等于`k`那么此时只能用`Bellman - ford`算法来做
3. 在算法问题中，我们只用考虑对问题的抽象建图过程，以及对这些算法的应用过程，而不考虑这些算法的证明
4. 稀疏图用邻接矩阵存储，稠密图用邻接矩阵存储
5. 在最短路算法中，无向图看作是一种特殊的有向图，所以我们只用考虑有向图的做法，无向图相当于一条边存两个方向边，有向图的做法可以解决无向图的问题

邻接矩阵的图的存储：

```
//用一个二维数组来存储边的数据
g[a][b] = c;//表示从a -> b这条边的权重是c
```

朴素Dijkstra

基本思想

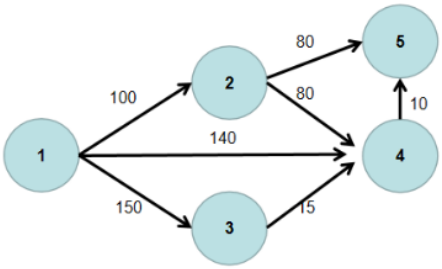
这里引用 `acwing` 用户 `Hasity` 的题解：

迪杰斯特拉算法采用的是一种贪心的策略。

求源点到其余各点的最短距离步骤如下：

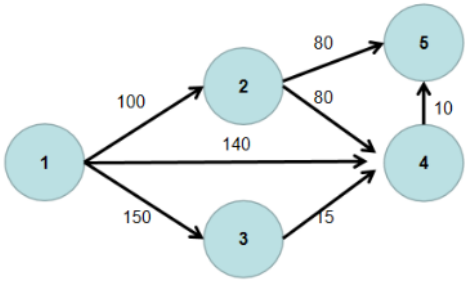
1. 用一个 `dist` 数组保存源点到其余各个节点的距离，`dist[i]` 表示源点到节点 `i` 的距离。初始时，`dist` 数组的各个元素为无穷大。
   用一个状态数组 `state` 记录是否找到了源点到该节点的最短距离，`state[i]` 如果为真，则表示找到了源点到节点 `i` 的最短距离，`state[i]` 如果为假，则表示源点到节点 `i` 的最短距离还没有找到。初始时，`state` 各个元素为假。

序号	dist	state
1	∞	0
2	∞	0
3	∞	0
4	∞	0
5	∞	0



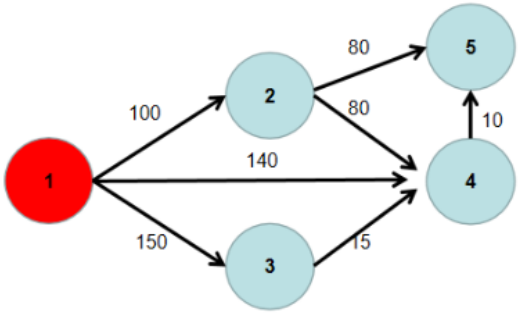
2. 源点到源点的距离为 `0`，即 `dist[1] = 0`

序号	dist	state
1	0	0
2	∞	0
3	∞	0
4	∞	0
5	∞	0



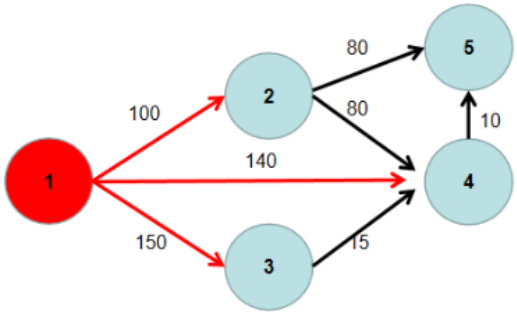
3. 遍历 `dist` 数组，找到一个节点，这个节点是：没有确定最短路径的节点中距离源点最近的点。假设该节点编号为 `i`。此时就找到了源点到该节点的最短距离，`state[i]` 置为 `1`，如下图，最开始的时候没有确定最短路径的节点中距离源点最近的点是 `1`，我们将 `state[i]` 置为 `1`，表示找到了源点到该节点的最短距离

序号	dist	state
1	0	1
2	∞	0
3	∞	0
4	∞	0
5	∞	0



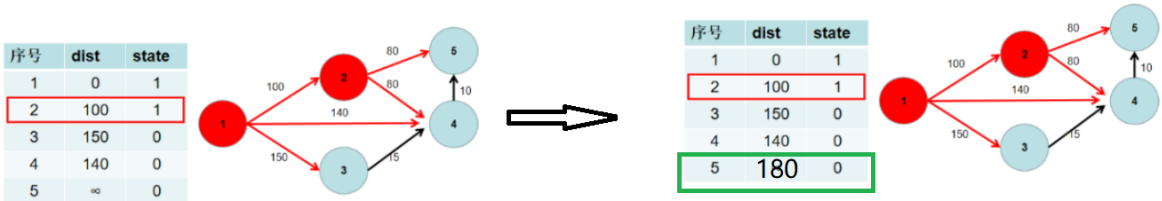
4. 遍历 `i` 所有可以到达的节点 `j`，如果 `dist[j]` 大于 `dist[i]` 加上 `i -> j` 的距离，即 `dist[j] > dist[i] + w[i][j]` （`w[i][j]` 为 `i -> j` 的距离），则更新 `dist[j] = dist[i] + w[i][j]`，如下图，`1` 可以到 `2 3 4`，并且距离小于原来的距离，更新这些点的距离

序号	dist	state
1	0	1
2	100	0
3	150	0
4	140	0
5	∞	0



5. 重复 3 4 步骤，直到所有节点的状态都被置为 `1`

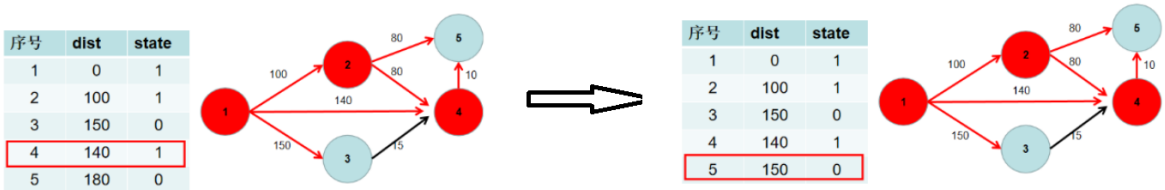
选中 `2` 号点，并更新 `5` 号点的距离：



序号	dist	state
1	0	1
2	100	1
3	150	0
4	140	0
5	∞	0

序号	dist	state
1	0	1
2	100	1
3	150	0
4	140	0
5	180	0

选中 `4` 号点，并更新 `5` 号点的距离：

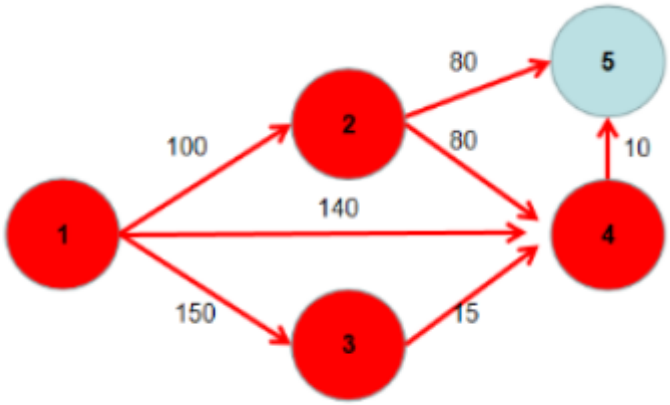


序号	dist	state
1	0	1
2	100	1
3	150	0
4	140	1
5	180	0

序号	dist	state
1	0	1
2	100	1
3	150	0
4	140	1
5	150	0

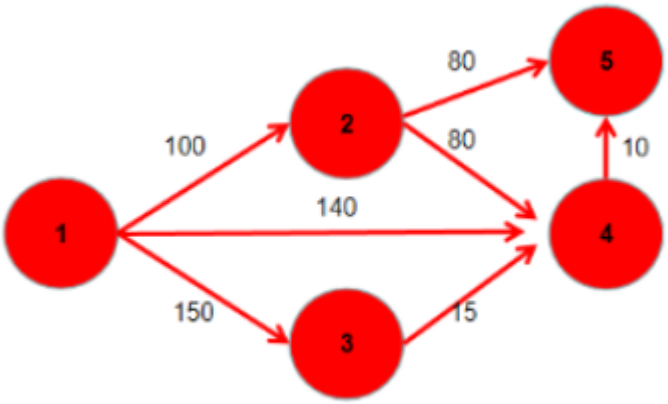
选中 `3` 号点，此时不更新距离：

序号	dist	state
1	0	1
2	100	1
3	150	1
4	140	1
5	150	0



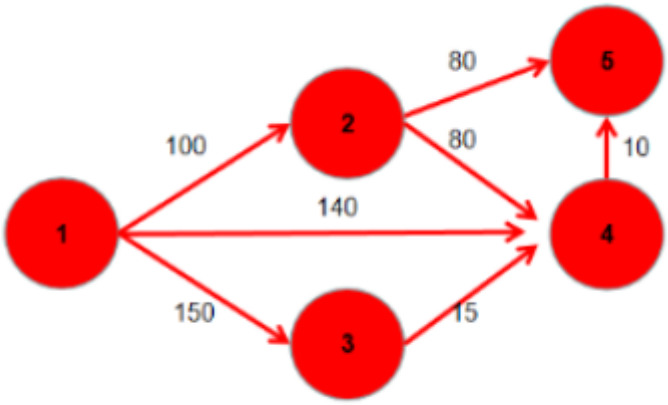
选中`5`号点，不更新距离：

序号	dist	state
1	0	1
2	100	1
3	150	1
4	140	1
5	150	1



最后每个点的最短距离就求出来了：

序号	dist	state
1	0	1
2	100	1
3	150	1
4	140	1
5	150	1



6. 此时`dist`数组中，就保存了源点到其余各个节点的最短距离

伪代码：

```
int dist[n],state[n];
dist[1] = 0, state[1] = 1;
//state[i] = 1表示点i进入了集合S中，表示这个点的最短距离已经找到
//state[i] = 0表示点i仍在集合T中，表示没有确定最短路径的节点中距离源点最近的点
for(i:1 ~ n)
{
    t <- 没有确定最短路径的节点中距离源点最近的点;
    state[t] = 1;
    更新 dist;
}
```

代码实现

给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环，所有边权均为正值。

请你求出 1 号点到 n 号点的最短距离，如果无法从 1 号点走到 n 号点，则输出 -1。

输入格式

第一行包含整数 n 和 m。

接下来 m 行每行包含三个整数 x, y, z，表示存在一条从点 x 到点 y 的有向边，边长为 z。

输出格式

输出一个整数，表示 1 号点到 n 号点的最短距离。

如果路径不存在，则输出 -1。

数据范围

$1 \leq n \leq 500$,
$1 \leq m \leq 10^5$,
图中涉及边长均不超过10000。

输入样例：

```
3 3
1 2 2
2 3 1
1 3 4
```

输出样例：

```
3
```

```
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;

const int N = 510;

int n, m;
//邻接矩阵，用来存放顶点
int g[N][N];
```

```cpp
//dist用来存放每个点到源点的最短距离
int dist[N];
//st[]用来表示第i个点的距离是否已经更新为最短距离
bool st[N];


int dijkstra()
{
    //首先，将所有的距离初始化为正无穷
    memset(dist, 0x3f, sizeof dist);
    //1号点的距离初始化为0
    //如果1号点的距离一开始不为0则所有点的dist都是无穷
    //更改距离dist[j] > dist[t] + g[t][j]就没有了意义
    dist[1] = 0;

    //迭代n次
    //每一次都确定一个点的最短距离
    for(int i = 0; i < n; i ++)
    {
        //每次先找到T集合中到源点距离最小的那一个
        //先将t赋值为-1表示没有找到
        int t = -1;

        //遍历所有的点开始寻找T集合中到源点距离最小的那一个
        for(int j = 1; j <= n; j ++)
        {
            //如果在T集合中&&
            //(找到了T集合中的第一个点(此时t为-1，这时当然要赋给t集合中第一个点)或者找到了T集合中的点但是比上一个点的距离小)
            //这个距离更小的点赋予t
            if(!st[j] && (t == -1 || dist[t] > dist[j])) t = j;
        }//出for循环的时候就找到了T集合中的距离最小的点

        //将这个点加入到S集合中
        st[t] = true;

        //用t来更新其他点的距离
        //遍历所有的点
        //如果是t可以到达的点
        //那么就按照dist[t] + g[t][j] < dist[j]正常更新
        //如果是t不可以到达的点，则g[t][j] = 0x3f3f3f3f
        //所以dist[t] + g[t][j]必然 > dist[j]，dist[j]必然不会更新
        //所以我们可以统一遍历所有的点即可
        for(int j = 1; j <= n; j ++)
        {
            //对于自环，其实没有影响
            //我们在更新dist[]数组的过程中必然不会加上自环
            dist[j] = min(dist[j], dist[t] + g[t][j]);
           // cout << j << " : " << dist[j] << endl;
        }


```

```
    }
    //如果n的最短距离仍然是无穷，说明不连通
    if(dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}



int main()
{
    //读入点数和边数
    scanf("%d%d", &n ,&m);

    //邻接矩阵的初始化
    memset(g, 0x3f, sizeof g);

    //读入所有的边
    while(m --)
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        //如果有重边，则只读入最小的那条边
        g[a][b] = min(g[a][b], c);
    }

    int t = dijkstra();

    //输出从1号点到n号点的最短距离
    printf("%d\n", t);

    return 0;

}
```

去掉注释后代码行数不超过50行，简洁优雅到了极致。。。

1. **关于重边的问题**

读入边的时候就直接读入了最短的那条边，直接忽略了重边

2. **关于自环的问题**

首先我们要知道最短路径必然不会包含自环这个边，所以我们在算法中更新其他所有的点的时候`dist[j] = min(dist[j], dist[t] + g[t][j])`，如果是自己，则必然还是`dist[t] < dist[t] + g[t][t]`，不会更新加上自环，如果是其他的点，显然`g[t][j]`不包含自环

堆优化Dijkstra

基本思想

如果给定的图是一个稀疏图，顶点的个数达到了`10 ^ 5`的级别，那么这时我们再使用朴素做法两重`for`循环时间复杂度就会很大

所以需要进行优化

我们先看看朴素`Dijkstra`中的核心代码：

```
for(int i = 0; i < n; i ++)
{
    int t = -1;
    for(int j = 1; j <= n; j ++)
    {
        if(!st[j] && (t == -1 || dist[j] < dist[t])) t = j;
    }
    st[t] = true;
    for(int j = 1; j <= n; j ++)
    {
        dist[j] = min(dist[j], dist[t] + g[t][j]);
    }
}
```

最外层需要循环`n`次，在这`n`次循环中

第一个`for`循环不必多说，它总共执行了`n ^ 2`次

第二个`for`循环我们需要注意，虽然看起来在一次外循环中也循环了`n`次，但是在某一次外循环中，此时用`t`去更新所有的边的时候，只有当`t`和`j`之间存在一条边的时候赋值语句才可能会有效执行，因为如果不存在一条边，那`g[t][j] = 0x3f3f3f3f`，显然`dsit[j]`还是保持原来的值，所以在总的外循环中赋值语句可能执行的最大次数其实是`m`次，也就是边的个数

也可以这样理解，这个`for`循环是`n ^ 2`但是因为是稠密图和`m`一个级别的也就是`m`

所以当给定的图是稀疏图的时候，边的个数级别和顶点的个数级别相同，我们需要优化的就只是第一个`for`循环

代码实现

给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环，所有边权均为非负值。

请你求出 1 号点到 n 号点的最短距离，如果无法从 1 号点走到 n 号点，则输出 -1。

输入格式

第一行包含整数 n 和 m。

接下来 m 行每行包含三个整数 x, y, z，表示存在一条从点 x 到点 y 的有向边，边长为 z。

输出格式

输出一个整数，表示 1 号点到 n 号点的最短距离。

如果路径不存在，则输出 -1。

数据范围

$1 \leq n, m \leq 1.5 \times 10^5$，
图中涉及边长均不小于 0，且不超过 10000。
数据保证：如果最短路存在，则最短路的长度不超过 10^9。

输入样例：

```
3 3
1 2 2
2 3 1
1 3 4
```

输出样例：

```
3
```

```cpp
#include<iostream>
#include<cstring>
#include<algorithm>
#include<queue>
#include<vector>
using namespace std;
const int N = 150010;
typedef pair<int, int> PII;
int n, m;
//邻接表中新增一个数组w[]用来存权重
int h[N], e[N], ne[N], w[N], idx;
//dist存放每个点到源点的最短距离
int dist[N];
//st[]用来指示这个点是否更新成了最短距离
//st[i] = true表示这个点的dist[]更新成了最短距离，这个点加入到S集合中
bool st[N];

//加入一条边，这里还需要加入一条权重
void add(int a, int b, int c)
{
    w[idx] = c;
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx;
```

```
    idx ++;
}

int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    //定义一个小根堆
    //堆里面是结点编号和距离，所以堆中的元素是pair
    //后面两个参数表明这是一个小根堆
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    //将第一个点放入堆中，距离是0，编号是1
    heap.push({0, 1});

    //当堆不为空的时候
    while(heap.size())
    {
        //找到当前距离源点最近的点
        auto t = heap.top();
        heap.pop();

        //ver是顶点编号
        //distance是这时对应的顶点的最短距离
        int ver = t.second, distance = t.first;

        if(st[ver]) continue;
        //由于我们每次用这个顶点去更新其他点的时候都会向堆中加入新的点，所以需要判重
        //注意下面一行代码也可以不用写，当用重复的点去更新的时候是始终不满足dist[j] > distance + w[i]这个条件的
        //因为重复的点的distance显然是大于前面第一次遍历的这个点的distance
        //所以不会对它的邻接点的距离有任何改动
        st[ver] = true;

        //遍历这个点可以到达的所有点，更新这些点的距离
        //注意在朴素版本的中我们遍历的是所有的点
        //这里我们遍历的是所有的边
        for(int i = h[ver]; i != -1; i = ne[i])
        {
            //
            int  j = e[i];
            //如果当前j这个点的距离大于从t点过来的距离
            //我们就进行更新
            if(dist[j] > distance + w[i])
            {
                //更新距离
                dist[j] = distance + w[i];
                //把顶点j以及最新的距离加入到堆中
                //注意到j以及它之前的距离仍然在堆中我们并没有删除
                //但是没有没有影响，因为每次从堆中取出的都是最小的距离以及对应的顶点
                //所以在最坏的情况下每遍历一条边我们会往堆中加入1个元素
                //最坏情况下堆中有m个元素
                //所以使用vector更新一次边的时间复杂度是O(logm)
```

```
                heap.push({dist[j], j});
            }
        }
    }
    if(dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}


int main()
{
    cin >> n >> m;
    memset(h, -1, sizeof h);

    //读入m条边
    while(m --)
    {
        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, c);
    }

    cout << dijkstra() << endl;

    return 0;

}
```

重点来分析一下以下几个问题：

1. **重边和自环怎么解决?**

我们注意下面这段代码：

```
if(st[ver]) continue;
st[ver] = true;
```

当`st[ver] = true`的时候就跳过循环继续`continue`，为什么会这样呢?

我们看这一段：

```
if(dist[j] > distance + w[i])
{
    dist[j] = distance + w[i];
    heap.push({dist[j], j});
}
```

注意到假如当前确定的最短距离顶点`t`的出边对应的顶点`j`中存在重边，`t -> j`的权重分别为`2 1`，假设`dist[t] = 0`遍历`t`的的所有边的时候，第一次`for`循环，遍历到了顶点`j`，满足`dist[j] > distance + w[i]`，更新`dist[j] = 2`，然后将`{2, j}`加入到堆中，第二次`for`循环，又遍历到了这个边，此时边的权重变为`1`，仍然满足`dist[j] > distance + w[i]`，更新`dsit[j] = 1`，然后将`{1, j}`加入到堆中

所以我们看到，有重边的情况下，在堆中可以存在顶点相同的多个`距离 - 顶点`对，在下一次找当前距离源点最短距离的点的时候可能会找到`{1, j}`，但是再下一次`{2, j}`可能仍是堆中的距离最短`距离 - 顶点`对，但是此时我们已经更新了顶点`j`，`j`已经属于集合`s`，不应该再给`t`赋值`j`，所以此时需要跳过

所以对于重边，要么不满足`dist[j] > distance + w[i]`不会加入到堆中，要么是在堆中会因为`st[]`数组的标记而跳过

对于自环，自环必然不满足`dist[j] > distance + w[i]`，所以自环必然不会进入到堆中，不会进行处理

当然堆中出现重复的`距离 - 顶点`对并不仅仅是重环导致的，只要顶点的距离发生了更新，就会加入重复的`距离 - 顶点`对

2. **时间复杂度是多少?**

在稀疏图中我们看下面这段伪码

```
1 for(i:1 ~ n)
2 {
3     t <- 没有确定最短路径的节点中距离源点最近的点;
4     state[t] = 1;
5     更新 dist;
6 }
```

上面第`3`行代码在每次循环的过程中的复杂度是`O(1)`（堆操作），所以在`n`次循环的时间复杂度是`O(n)`

上面的第`5`行代码，在堆优化版本的代码中，每次循环都是遍历当前顶点`t`的出边，在所有的`for`循环中，我们遍历的是所有的边，也就是遍历了`m`次，而每次遍历一条边在最坏的情况下都需要加入一个`距离 - 顶点`对到堆中，时间复杂度是`O(n)`，`m`次总共是`O(mlogn)`

所以总的时间复杂度就是`O(mlogn)`

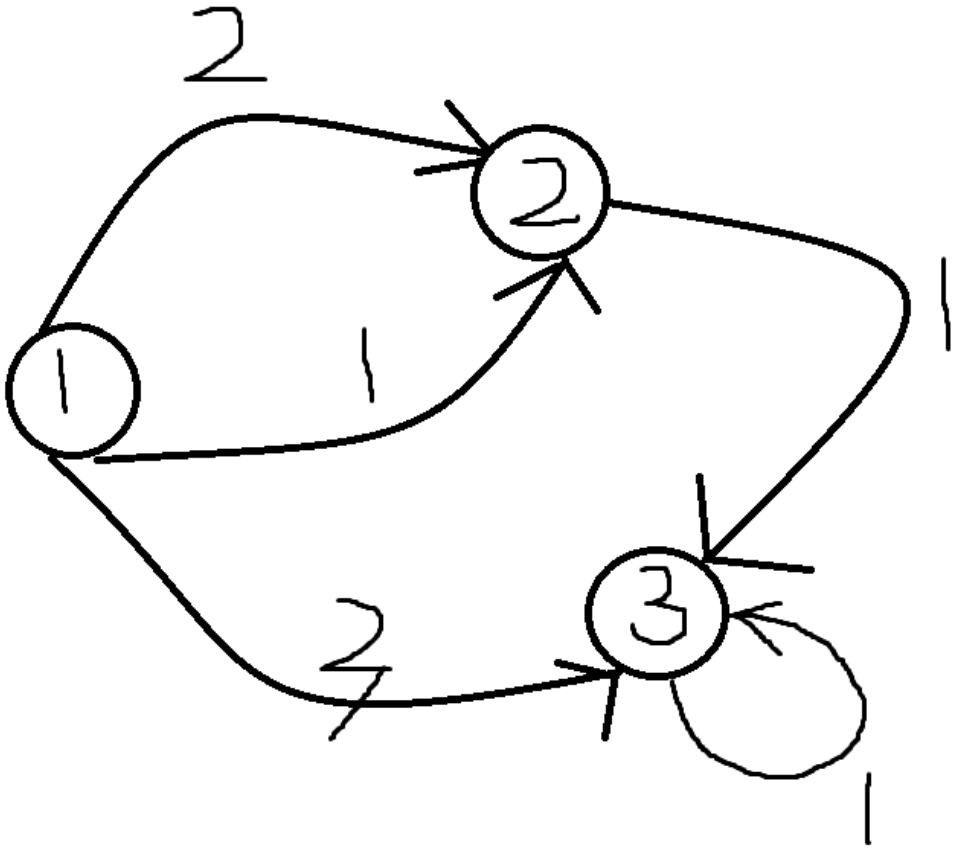
堆的写法有两种

1. 手写堆，这种方式可以保证堆中时时刻刻只有`n`个数，因为我们可以任意删除某个冗余的`距离 - 顶点`对其时间复杂度是标准的`O(mlogn)`
2. `stl`中的优先队列，这时我们不能任意删除某个冗余的`距离 - 顶点`对，
    1. 在最坏的情况下每遍历一条边我们会往堆中加入`1`个元素，最坏情况下堆中有`m`个元素，所以使用`vector`更新一次边的时间复杂度是`O(logm)`，总的时间复杂度是`O(mlogm)`

但是我们需要注意的是这是一个稀疏图，`n`的数量级跟`m`是相似的，其实总的时间复杂度还是`O(mlogm)`

3. **堆清空的情况是什么，换句话说，过程是什么?**

我们以下图为例来分析过程：



一开始，读入所有的边，包括自环和重边

将`dist[1] = 0`后加入`{0, 1}`入堆中

进入`while`循环

取出顶点`1`，并弹出，然后遍历它的所有的邻接顶点，加入顶点对`{2, 2}, {1, 2}, {3, 3}`入堆中

此时堆中的顶点对为`{2, 2}, {1, 2}, {3, 3}`

下一次`while`循环

取出顶点`2`，并弹出，标记，然后遍历它的所有邻接顶点，加入`{2, 3}`入堆中

此时堆中的顶点对为`{2, 2}, {3, 3}, {2, 3}`

下一次`while`循环

取出顶点`2`，并弹出，发现`2`已经被标记过了属于集合`S`的顶点，跳过本次`while`循环

此时堆中的顶点对为`{3, 3}, {2, 3}`

下一次`while`循环

取出顶点`3`，并弹出，标记，然后遍历它的所有邻接顶点，自环直接不满足条件`dist[3] > dist[3] = distance`，不向堆中加入任何元素

此时堆中的顶点对为`{3, 3}`

下一次`while`循环

取出顶点`3`，并弹出，发现`3`已经被标记过了属于集合`S`的顶点，跳过本次`while`循环

此时堆中的顶点堆为空

跳出`while`循环

得到`dist[3] = 2`，返回