目录
导航

thousfeet

Try to learn something about everything and everything about something.

看完就懂了！一篇搞定图论最短路径问题

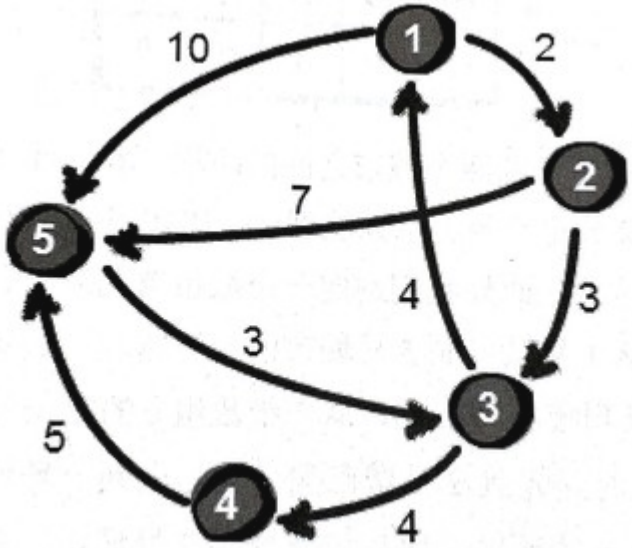
最最原始的问题——两点间的最短路

这类背景一般是类似：已知各城市之间距离，请给出从城市A到城市B的最短行车方案 or 各城市距离一致，给出需要最少中转方案。

也就是，**固定起始点的情况下，求最短路**。

这个问题用简单的搜索就能轻松解决。（本部分内容不涉及图论算法，可跳过）

假设用邻接矩阵存图，就比如下面这个例子：



	1	2	3	4	5
1	0	2	∞	∞	10
2	∞	0	3	∞	7
3	4	∞	0	4	∞
4	∞	∞	∞	0	5
5	∞	∞	3	∞	0

深度优先搜索（dfs）的做法：

```
void dfs(int cur, int dis) //cur-当前所在城市编号，dis-当前已走过的路径
{
```

```
    if(dis > min) return; //若当前路径已比之前找到的最短路大，没必要继续尝试（一个小优化，可
    if(cur == n) //当前已到达目的城市，更新min
    {
        if(dis < min) min = dis;
        return;
    }

    for(int i = 1; i <= n; i++) //对1~n号城市依次尝试
    {
        if(e[cur][i] != INF && book[i] == 0) //若cur与i可达，且i没有在已走过的路径中
        {
            book[i] = 1; //标记i为已在路径中
            dfs(i, dis+e[cur][i]); //继续搜索
            book[i] = 0; //对从i出发的路径探索完毕，取消标记
        }
    }
}
```

顺带插播一下如何理解DFS算法，它的关键思想仅在于解决**当下该如何做**。至于“下一步如何做”则与“当下该如何做”是一样的，把参数改为进入下一步的值再调用一下dfs()即可。

而在写dfs函数的时候就**只要解决当在第step的时候你该怎么办**，通常就是把每一种可能都去尝试一遍。当前这一步解决后便进入下一步dfs(step+1)，剩下的事情就不用管它了。

基本模型：

```
void dfs(int step)
{
    判断边界
    尝试每一种可能 for(int i = 1; i <= n; i++)
    {
        继续下一步 dfs(step+1)
    }
}
```

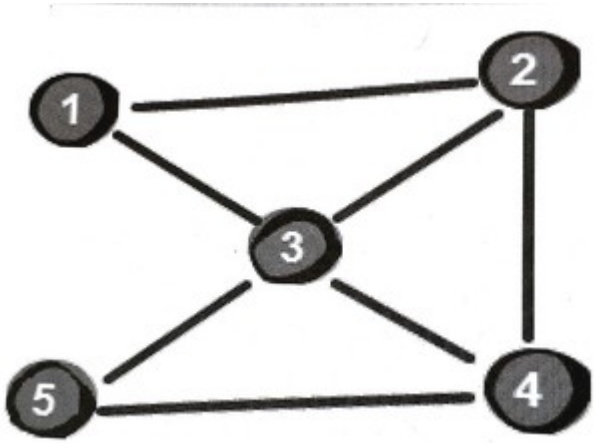
目录
导航

1. 超详细！CentOS 7 + Hadoop3.0.0 搭建伪分布式集群(43969)
2. 看完就懂了！一篇搞定图论最短路径问题(36538)
3. 超详细！Github团队协作教程（Gitkraken版）(35818)
4. 在Linux服务器上运行Jupyter notebook server教程(23066)
5. 2018计算机专业保研面经（清华、浙大、华科）(12363)

评论排行榜

1. 2018计算机专业保研面经（清华、浙大、华科）(22)
2. 《面向对象程序设计》第四次作业（考虑优先级的表达式计算）(12)
3. Alpha冲刺！ Day9 - 砍柴(9)
4. 《软工实践》团队项目 - 团队展示(9)
5. 第二次作业电梯编程题测试结果(8)

但对于所有**边权相同**的情况，用广度优先搜索会更快更方便。



比如上面提到的最少中转方案问题，问从城市1到城市4需要经过的最少中转城市个数。

用广搜的做法：

```
int bfs()
{
    queue<pair<int,int>> que; //pair记录城市编号和dis，也可以用结构体
    que.push({1,0}); //把起始点加入队列
    book[1] = 1; //标记为已在路径中
    while(!que.empty())
    {
        int cur = que.front();
        que.pop();
        for(int i = 1; i <= n; i++)
        {
            if(e[cur][i] != MAX && book[i] == 0) //若从cur到i可达且i不在队列中，i入队
            {
                que.push({i, cur.second+1});
                book[i] = 1;
```

目录导航

17 0

```
                if(i == n) return cur.second; //如果已扩展出目标结点了，返回中转城市数答
            }
        }
    }
}
```

以上都是开胃，下面才是真的重点来了~

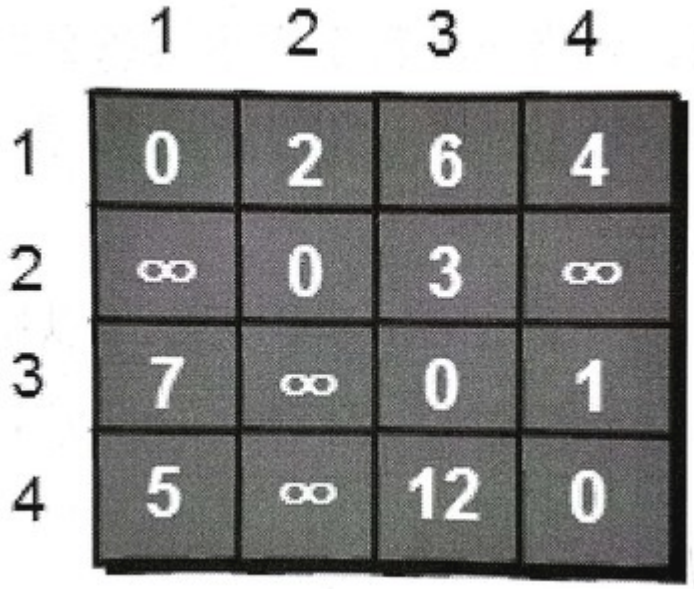
膨胀——任意两点间的最短路

已经知道了求解固定两点间的最短路，那要怎么求任意两点间的最短路呢？显然，可以进行n^2次的dfs或bfs轻松搞定（被打）。

观察会发现，如果要让两点 i , j 间的路程变短，只能通过第三个点 k 的中转。比如上面第一张图，从 1->5 距离为10，但 1->2->5 距离变成9了。事实上，**每个顶点都有可能使另外两个顶点间的路程变短**。这种通过中转变短的操作叫做松弛。

当任意两点间不允许经过第三个点时，这些城市之间的最短路程就是初始路程：

目录
导航



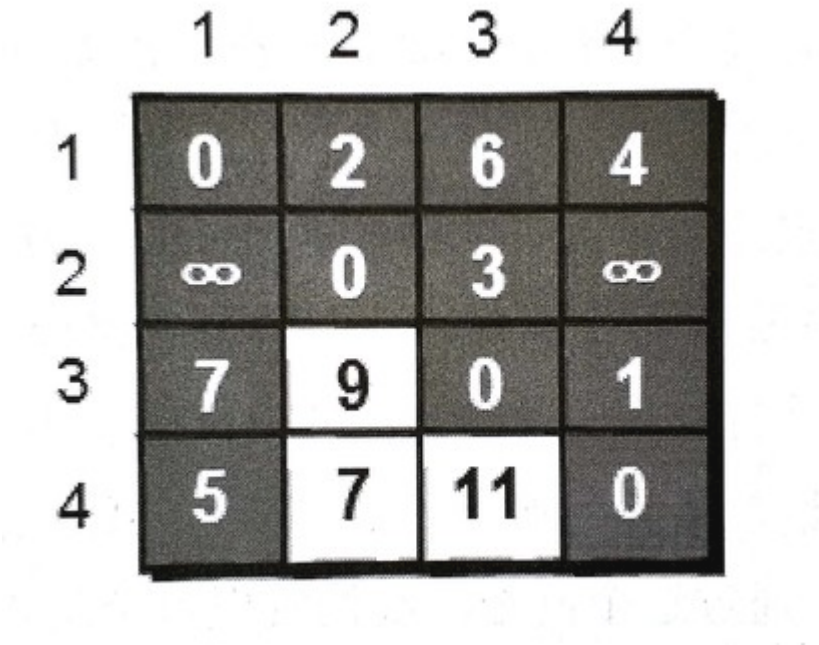
目录
导航

假如现在允许经过1号顶点的中转，求任意两点间的最短路，这时候就可以遍历每一对顶点，试试看通过1号能不能缩短他们的距离。

```
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= n; j++)
    {
        if(e[i][j] > e[i][1]+e[1][j]) e[i][j] = e[i][1]+e[1][j];
    }
```

更新后果然有好几条变短了：

17 0



扩展一下，先允许1号顶点作为中转给所有两两松弛一波，再允许2号、3号...n号都做一遍，就能得到最终任意两点间的最短路了。

这就是**Floyd算法**，虽然时间复杂度是令人发怵的O(n^3)，但核心代码只有五行，实现起来非常容易。

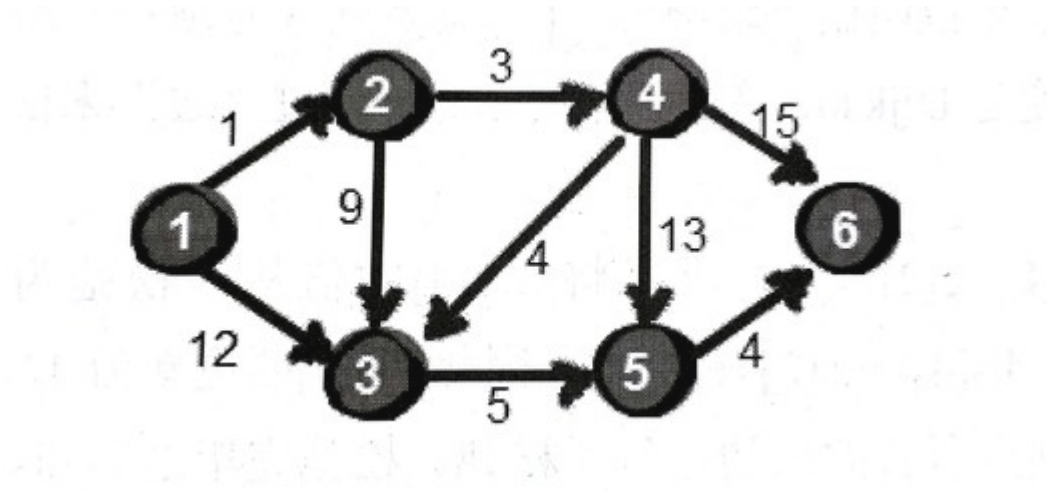
```
for(int k = 1; k <= n; k++)
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            if(e[i][j] > e[i][k]+e[k][j])
                e[i][j] = e[i][k]+e[k][j];
```

最常见的问题——单源最短路

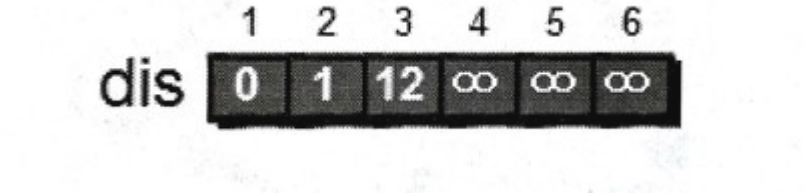
传说中如雷贯耳的“单源最短路”应该是做题中最常见到的问题了。也即，**指定源点，求它到其余各个结点的最短路。**

目录
导航

比如给出这张图，假设把1号结点作为源点。



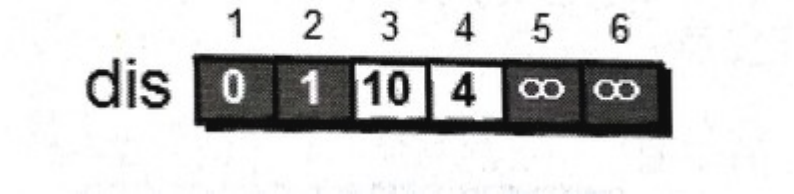
还是用数组dis来存1号到其余各点的初始路程：



既然是求最短路径，那先选一个离1号最近的结点，也就是2号结点。这时候，dis[2]=1 就固定了，它就是1到2的最短路径。这是为啥？因为目前离1号最近的是2号，且这个图的所有边都是正数，那就不可能能通过第三个结点中转使得距离进一步缩短了。因为从1号出发已经找不到哪条路比直接到达2号更短了。

选好了2号结点，现在看看2号的出边，有2->3和2->4。先讨论通过2->3这条边能否让1号到3号的路程变短，也即比较dis[3]和dis[2]+e[2][3]的大小。发现是可以的，于是dis[3]从12变为新的更短路10。同理，通过2->4也条边也更新下dis[4]。

松弛完毕后dis数组变为：



接下来，继续在剩下的 3 4 5 6 结点中选一个离1号最近的结点。发现当前是4号离1号最近，于是dis[4]确定了下来，然后继续对4的所有出边看看能不能做松弛。

balabala，这样一直做下去直到已经没有“剩下的”结点，算法结束。

这就是**Dijkstra算法**，整个算法的基本步骤是：

> 1. 所有结点分为两部分：已确定最短路的结点集合P、未知最短路的结点集合Q。最开始，P中只有源点这一个结点。（可用一个book数组来维护是否在P中）
> 2. 在Q中选取一个离源点最近的结点u（dis[u]最小）加入集合P。然后考察u的所有出边，做松弛操作。
> 3. 重复第二步，直到集合Q为空。最终dis数组的值就是源点到所有顶点的最短路。

代码：

```
for(int i = 1; i <= n; i++) dis[i] = e[1][i]; //初始化dis为源点到各点的距离
for(int i = 1; i <= n; i++) book[i] = 0;
book[1] = 1; //初始时P集合中只有源点

for(int i = 1; i <= n-1; i++) //做n-1遍就能把Q遍历空
{
    int min = INF;
    int u;
    for(int j = 1; j <= n; j++) //寻找Q中最近的结点
    {
        if(book[j] == 0 && dis[j] < min)
        {
            min = dis[j];
```

目录导航

17 0

```
            u = j;
        }
    }
    book[u] = 1; //加入到P集合
    for(int v = 1; v <= n; v++) //对u的所有出边进行松弛
    {
        if(e[u][v] < INF)
        {
            if(dis[v] > dis[u] + e[u][v])
                dis[v] = dis[u] + e[u][v];
        }
    }
}
```

Dijkstra是一种基于贪心策略的算法。每次新扩展一个路径最短的点，更新与它相邻的所有点。当所有边权为正时，由于不会存在一个路程更短的没扩展过的点，所以这个点的路程就确定下来了，这保证了算法的正确性。

但也正因为这样，这个算法**不能处理负权边**，因为扩展到负权边的时候会产生更短的路径，有可能破坏了已经更新的点路程不会改变的性质。

于是，**Bellman-Ford算法**华丽丽的出场啦。它不仅可以处理负权边，而且算法思想优美，且核心代码只有短短四行。

（用三个数组存边，第i条边表示u[i]->v[i]，权值为w[i]）

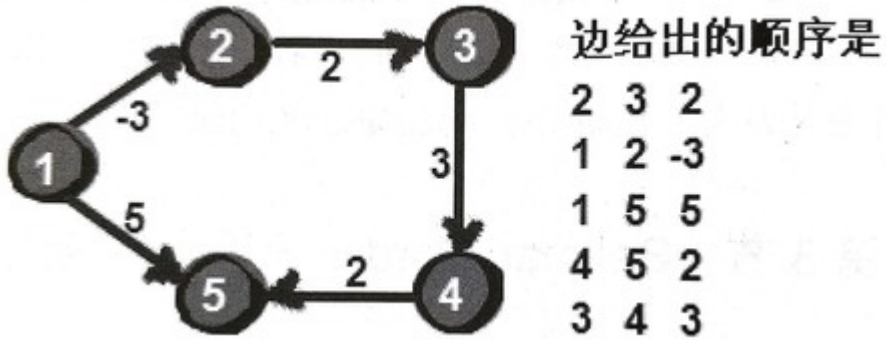
```
for(int k = 1; k <= n-1; k++)
    for(int i = 1; i <= m; i++)
        if(dis[v[i]] > dis[u[i]] + w[i])
            dis[v[i]] = dis[u[i]] + w[i];
```

后两行代码的意思是，看看能否通过u[i]->v[i]这条边缩短dis[v[i]]。加上第二行的for，也就是把所有的m条边一个个拎出来，看看能不能缩短dis[v[i]]（松弛）。

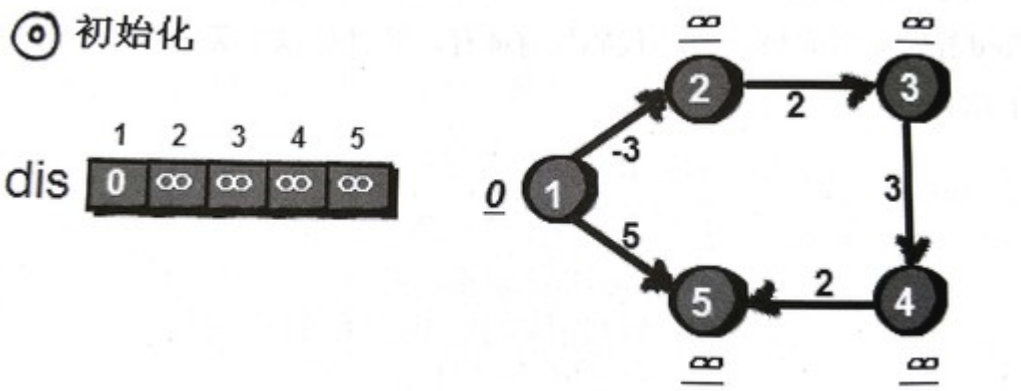
那把每一条边松弛一遍后有什么效果呢?

比如求这个例子：

目录
导航



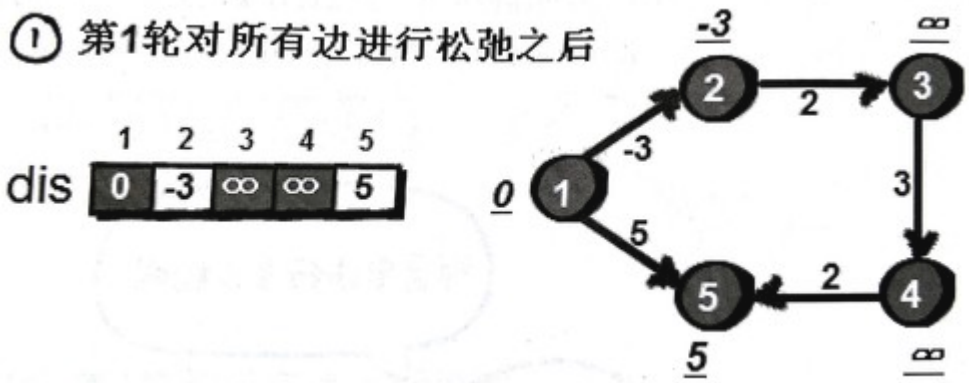
同样用dis数组来存储1号到各结点的距离。一开始时只有dis[1]=0，其他初始化为INF。



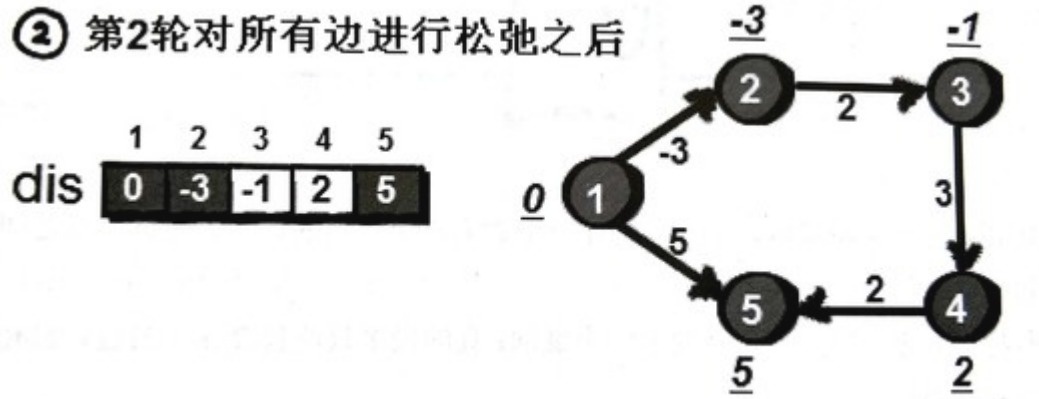
先来处理第一条边 2->3 ，然鹅dis[3]是INF，dis[2]+2也是INF，松弛失败。
第二条边 1->2 ，dis[2]是INF，dis[1]-3是-3，松弛成功，dis[2]更新为-3。

就这样对所有边松弛一遍后的结果如下：

目录
导航



这时候dis[2]和dis[5]的值变小了，如果再做一轮松弛操作的话，之前不成功的松弛这时候也能也就可以起作用了。



换句话说，第一轮松弛后得到的是从1号出发“只能经过1条边”到达其余各点的最短路，第二轮松弛后得到的是“只能经过2条边”到达其余各点的最短路，如果进行第k轮松弛得到的就是“只能经过k条边”到达其余各点的最短路。

那么到底需要进行多少轮呢？答案是**n-1轮**。因为在一个含有n个顶点的图中，任意两点间的最短路最多包含n-1条边。也就解释了代码的第一行，是在进行n-1轮松弛。

完整代码：

```
for(int i = 1; i <= n; i++) dis[i] = INF;
dis[1] = 0; //初始化dis数组，只有1号的距离为0

for(int k = 1; k <= n-1; k++) //进行n-1轮松弛
    for(int i = 1; i <= m; i++) //枚举每一条边
        if(dis[v[i]] > dis[u[i]] + w[i]) //尝试进行松弛
            dis[v[i]] = dis[u[i]] + w[i];
```

此外，Bellman-Ford算法还可以检测一个图是否含有负权回路。如果在进行了n-1次松弛之后，仍然存在某个 `dis[v[i]] > dis[u[i]] + w[i]` 的情况，还可以继续成功松弛，那么必然存在回路了（因为正常来讲**最短路径包含的边最多只会有n-1条**）。

判断负权回路也即在上面那段代码之后加上一行：

```
for(int i = 1; i <= m; i++)
    if(dis[v[i]] > dis[u[i]] + w[i]) flag = 1;
```

Bellman-Ford算法的时间复杂度是O(nm)，貌似比Dijkstra还高。事实上还可以进行优化，比如可以加一个bool变量check用来标记数组dis在本轮松弛中是否发生了变化，如果没有，就可以提前挑出循环。因为是“最多”达到n-1轮，实际情况下经常是早就已经达到最短，没法继续成功松弛了。

```
for(int k = 1; k <= n-1; k++) //进行n-1轮松弛
{
    bool check = 0;
    for(int i = 1; i <= m; i++) //枚举每一条边
        if(dis[v[i]] > dis[u[i]] + w[i]) //尝试进行松弛
        {
            dis[v[i]] = dis[u[i]] + w[i];
            check = 1;
        }
    if(check == 0) break;
}
```

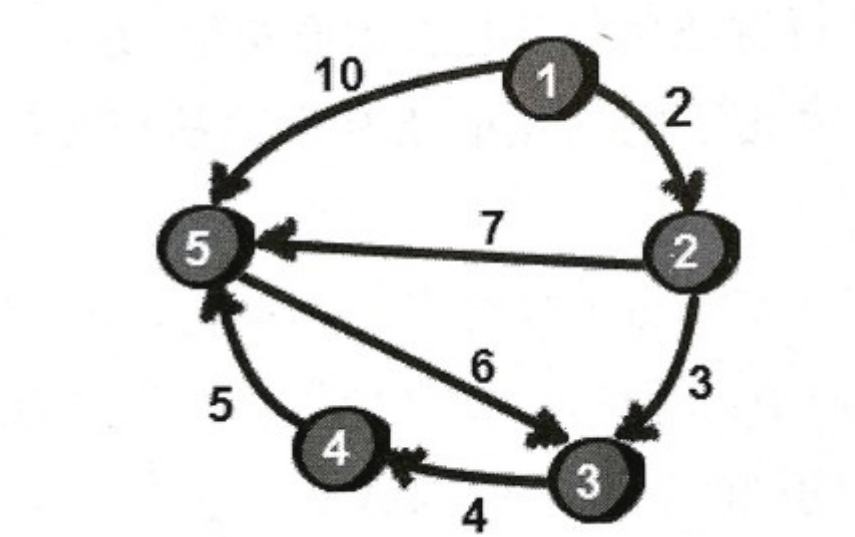
目录
导航

另外一种优化是：**每次仅对最短路估计值发生了变化的结点的所有出边进行松弛操作**。因为在上面的算法中，每实施一次松弛操作后，就会有一些顶点已经求得最短路之后便不会再改变了（由估计值变为确定值），既然都已经不受后续松弛操作的影响了却还是每次都要判断是否需要松弛，就浪费了时间。

可以用队列来维护dis发生了变化的那些结点。具体操作是：

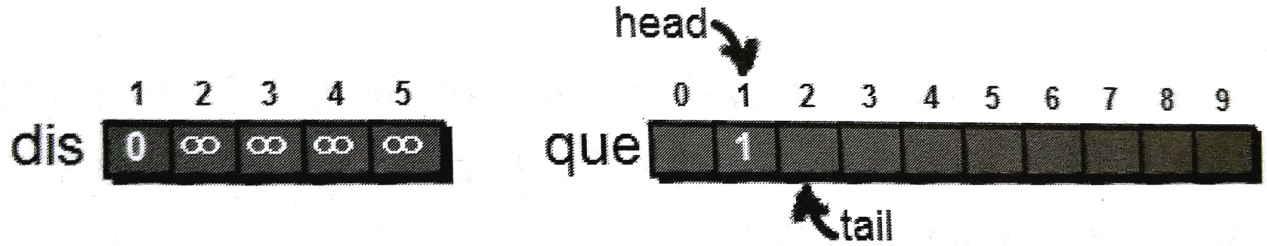
> 1. 初始时将源点加入队列。
> 2. 每次选取队首结点u，对u的所有出边进行松弛。假设有一条边u->v松弛成功了，那就把v加入队列。然而，同一个结点同时在队列中出现多次是毫无意义的（可以用一个bool数组来判哪些结点在队列中）。所以刚提到的操作其实是，如果v不在当前队列中，才把它加入队列。
> 3. 对u的所有出边松弛完毕后，u出队。接下来不断的取出新的队首做第2步操作，直到队列为空。

一个例子：

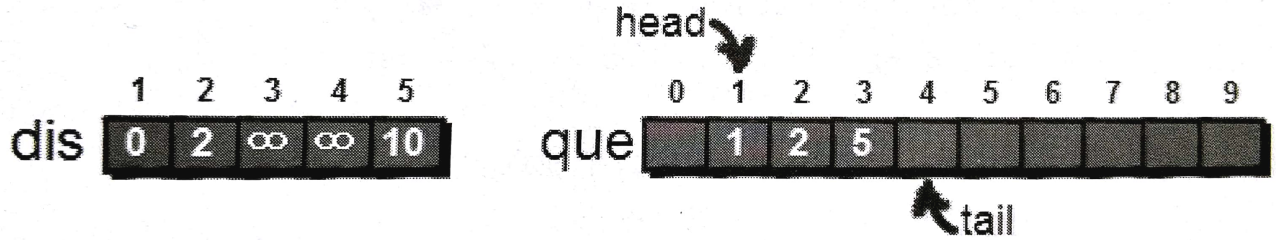


用数组dis来存放1号结点到各点的最短路。初始时dis[1]为0。接下来将1号结点入队。

目录
导航

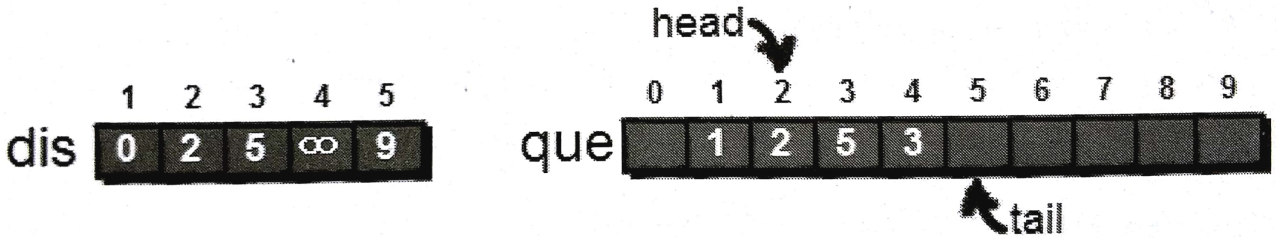


现在看1号的所有出边，对于1->2，比较dis[2]和dis[1]+e[1][2]的大小，发现松弛成功，dis[2]从INF变为2。并且2不在队列中，所以2号结点入队。同理，5号结点也松弛成功，入队。



1号结点处理完毕，此时将1号出队，接着对队首也就是2号结点进行同样的处理。在处理2->5这条边的时候，虽然松弛成功，dis[5]从10更新为9了，但5号顶点已经在队列中，所以5号不能再次入队。

处理完2号之后就长这样：



接着一直持续下去，直到队列为空，算法结束。

代码：

目录
导航

```
for(int i = 1; i <= n; i++) book[i] = 0; //初始时都不在队列中
queue<int> que;
que.push(1); //将结点1加入队列
book[1] = 1; //并打标记

while(!que.empty())
{
    int cur = que.empty(); //取出队首
    for(int i = 1; i <= n; i++)
    {
        if(e[cur][i] != INF && dis[i] > dis[cur]+e[cur][i]) //若cur到i有边且能够松弛
        {
            dis[i] = dis[cur]+e[cur][i]; //更新dis[i]
            if(book[i] == 0) //若i不在队列中则加入队列
            {
                que.push(i);
                book[i] = 1;
            }
        }
    }

    que.pop(); //队首出队
    book[cur] = 0;
}
```

这其实就是**SPFA算法**（队列优化的Bellman-Ford），它的关键思想就在于：只有那些在前一遍松弛中改变了最短路估计值的结点，才可能引起它们邻接点最短路估计值发生改变。

它也能够判断负权回路：如果某个点**进入队列的次数超过n次**，则存在负环。

最短路径算法的对比

目录
导航

	Floyd	Dijkstra	Bellman-Ford	队列优化的 Bellman-Ford
空间复杂度	$O(N^2)$	$O(M)$	$O(M)$	$O(M)$
时间复杂度	$O(N^3)$	$O((M+N)\log N)$	$O(NM)$	最坏也是 $O(NM)$
适用情况	稠密图 和顶点关系密切	稠密图 和顶点关系密切	稀疏图 和边关系密切	稀疏图 和边关系密切
负权	可以解决负权	不能解决负权	可以解决负权	可以解决负权
有负权边	可以处理	不能处理	可以处理	可以处理
判定是否存在负权回路	不能	不能	可以判定	可以判定

目录
导航

附

文中的图片和部分文字来自《啊哈！算法》，博文所做的是整理书的内容和自己的想法。

上面全都用邻接矩阵存图是因为注重算法本身，矩阵存图好理解。但平时打题的时候最爱用邻接表，几乎不用邻接矩阵，因为常见的是稀疏图，也即边数 m << n^2，用邻接表存节省复杂度。请看：图的存储结构之邻接表（详解）

分类： 算法&数构

thousfeet
关注 - 26
粉丝 - 159
+加关注