图论入门

雷康宁

西北工业大学附属中学

2025 年 8 月 24 日

前言

课程内容

- 最短路
- 最小生成树
- 树上问题
- DAG 相关

Bellman-Ford

SPFA 原型，简单介绍点到为止。

Bellman-Ford

SPFA 原型，简单介绍点到为止。

该算法主要应用到松弛操作。

显然地，最短路至多是将所有点连起来，所以最多只会有 $n-1$ 条边。

所以我们进行 $n-1$ 次松弛操作，每一次松弛操作遍历所有边即可。

所以，时间复杂度 $O(nm)$ ，结束。

另外，如果松弛次数超过了 $n-1$ 次，还可以进行松弛，那么说明图中出现了负环。

Bellman-Ford

SPFA 原型，简单介绍点到为止。

该算法主要应用到松弛操作。

显然地，最短路至多是将所有点连起来，所以最多只会有 $n-1$ 条边。

所以我们进行 $n-1$ 次松弛操作，每一次松弛操作遍历所有边即可。

所以，时间复杂度 $O(nm)$ ，结束。

另外，如果松弛次数超过了 $n-1$ 次，还可以进行松弛，那么说明图中出现了负环。

SPFA

我们注意到，在 Bellman-Ford 中，有一些多余的松弛操作，比如反复对距离为 INF 的点相关的边进行松弛，这是无用的。

SPFA

我们注意到，在 Bellman-Ford 中，有一些多余的松弛操作，比如反复对距离为 INF 的点相关的边进行松弛，这是无用的。

显然地，只有上一次被松弛的点连接的边，才有可能引起下一次的松弛操作。

SPFA

我们注意到，在 Bellman-Ford 中，有一些多余的松弛操作，比如反复对距离为 INF 的点相关的边进行松弛，这是无用的。

显然地，只有上一次被松弛的点连接的边，才有可能引起下一次的松弛操作。

所以，我们只需要每一次使用队列存储松弛的点，在下一次直接取出来松弛即可。

SPFA

我们注意到，在 Bellman-Ford 中，有一些多余的松弛操作，比如反复对距离为 INF 的点相关的边进行松弛，这是无用的。

显然地，只有上一次被松弛的点连接的边，才有可能引起下一次的松弛操作。

所以，我们只需要每一次使用队列存储松弛的点，在下一次直接取出来松弛即可。

但是这个东西日常被卡，即使加入各种如 **SLF**、**LLL** 优化之类，也很容易被出题人卡挂，所以如非必要不建议使用。

Dijkstra

Dijkstra 也是进行松弛操作的，但是 SPFA 是基于一条边进行松弛，而 Dijkstra 是基于一个点进行松弛。

Dijkstra

Dijkstra 也是进行松弛操作的，但是 SPFA 是基于一条边进行松弛，而 Dijkstra 是基于一个点进行松弛。

将节点分成两个集合，一个是已经确定最短路的，另一个是没有确定的。

1. 每一次从没有确定的集合中找出最短路最短的节点，放进确定的集合中.
2. 对于刚刚被加入点的出边，进行松弛。

Dijkstra

Dijkstra 也是进行松弛操作的，但是 SPFA 是基于一条边进行松弛，而 Dijkstra 是基于一个点进行松弛。

将节点分成两个集合，一个是已经确定最短路的，另一个是没有确定的。

1. 每一次从没有确定的集合中找出最短路最短的节点，放进确定的集合中.
2. 对于刚刚被加入点的出边，进行松弛。

朴素做法是每次暴力从未确定集合中取值，取一次要 $O(n)$，总共 $O(n)$ 次，所以操作 1 总复杂度是 $O(n^2)$ 的。

如果我们使用优先队列进行优化，可以达到 $O(m \log n)$。

关于正确性和时间复杂度的证明，这里就不证了，参见 OI-wiki。

Prim

Prim 虽然在稠密图和完全图上复杂度显得比 Kruskal 更优，但不一定跑得更快，因此不建议写。

Prim

Prim 虽然在稠密图和完全图上复杂度显得比 Kruskal 更优，但不一定跑得更快，因此不建议写。

算法思想是初始任选一个点，然后更新到距离最小的一个点，把路径加入生成树，然后用新点来更新距离，反复进行。

Kruskal

Kruskal 是一种贪心，将边先按照升序排序，然后贪心地从小到大加入边，如果一条边连接的两点本身就已经联通，则不加入。

Kruskal

Kruskal 是一种贪心，将边先按照升序排序，然后贪心地从小到大加入边，如果一条边连接的两点本身就已经联通，则不加入。

只需要维护并查集即可，使用 $O(m\alpha(n))$ 或者 $O(m\log n)$ 都可以，不会造成瓶颈。

（注：这里的 α 函数是阿克曼函数的反函数，增长很缓慢，基本可以当成常数）

总时间复杂度 $O(m\log m)$。

Boruvka

一个神秘好用的古老算法，最小生成树和最小生成森林都可以求。在 1926 年就已经出现，当时还不是计算机算法。

Boruvka

一个神秘好用的古老算法，最小生成树和最小生成森林都可以求。在 1926 年就已经出现，当时还不是计算机算法。

我们先引入定义：

定义集合 E' 为找到的最小生成森林的边集

定义连通块为一个点集，使得集合中任意两个点都可以互相到达

定义一个联通块的“最小边”为连向其他连通块的最短边

Boruvka

它的算法逻辑如下：

0. 初始定义每一个节点为一个连通块
1. 设置每个连通块为“无最小边”
2. 找跨块的边，并且记录到两边的连通块的最小边中
3. 如果每一个连通块都没有最小边，那么结束，否则使用所有记录的最小边将连通块连起来，也就是将这个边加入 E' 中
4. 不断重复 $1 \sim 4$，最终使得所有点联通

一般地，我们使用遍历每一条边的方法完成第二步，需要 $O(m)$，又需要合并 $O(\log n)$ 次，所以是 $O(m \log n)$ 的总复杂度。

Boruvka

它的算法逻辑如下：

0. 初始定义每一个节点为一个连通块
1. 设置每个连通块为“无最小边”
2. 找跨块的边，并且记录到两边的连通块的最小边中
3. 如果每一个连通块都没有最小边，那么结束，否则使用所有记录的最小边将连通块连起来，也就是将这个边加入 E' 中
4. 不断重复 $1 \sim 4$，最终使得所有点联通

一般地，我们使用遍历每一条边的方法完成第二步，需要 $O(m)$，又需要合并 $O(\log n)$ 次，所以是 $O(m \log n)$ 的总复杂度。

但在一些题目里有特殊性质，使得可以快速完成第二步，这时候 Boruvka 的优势就很大了。

DAG 与拓扑排序

DAG，即有向无环图。

DAG 与拓扑排序

DAG，即有向无环图。

拓扑排序，即为对有向无环图上的点排序的一种方法，使得每一条边的起始点在拓扑序上先于结尾点。

DAG 与拓扑排序

DAG，即有向无环图。

拓扑排序，即为对有向无环图上的点排序的一种方法，使得每一条边的起始点在拓扑序上先于结尾点。

DAG 一定可以拓扑排序，而不能拓扑排序的图一定不是 DAG。

Tarjan

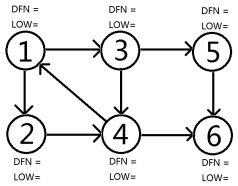
这里我们只介绍使用 Tarjan 求强连通分量的方法。

Tarjan

这里我们只介绍使用 Tarjan 求强连通分量的方法。

Tarjan 基于 DFS 算法。

在 DFS 过程中，我们对于每一个点记录 DFN 和 LOW 两个信息，同时把沿途的所有点都压进栈里。LOW 表示可以前进到的点的 DFN 的最小值。



当我们退出的过程中，如果发现一个点的 DFN 和 LOW 值相等，就说明当前点是一个强连通分量的根节点。

此时一直出栈直到这个点，这些点组成一个强连通分量。

2-SAT

SAT 是适定性（Satisfiability）问题的简称。一般形式为 k - 适定性问题，简称 k-SAT。而当 $k > 2$ 时该问题为 NP 完全的。所以我们只研究 $k = 2$ 的情况。

1. 我们将一个元素拆成两个点表示 bool 元素的两种情况，有向边表示若起点成立，则终点一定成立。

2. 当拆点建图后，如果一个元素拆出的两个点 u,v 。存在有向图上的路径 (u−>v)，则点 u 可以推出点 v ，点 u 非法，则点 v 合法。

3. 有向无环图的情况下，合法点的拓扑序比非法点大。

4. Tarjan 后同一元素拆点强连通分量编号小的点是合法点。

5. 如果一个元素拆成的两个点在同一个强连通分量里，即强连通分量编号相同，那么整个序列无解。

树链剖分

众所周知，树也是图，所以下一个内容是树（重）链剖分。

树链剖分

众所周知，树也是图，所以下一个内容是树（重）链剖分。

首先，需要明确树链剖分是干什么的。

树链剖分

众所周知，树也是图，所以下一个内容是树（重）链剖分。

首先，需要明确树链剖分是干什么的。

所谓树链剖分，就是将树剖成许多的重链，使得树上问题可以方便地转化成序列上的问题，然后这个序列就可以直接或者使用其他的数据结构来间接地解决问题。

树链剖分

我们将树上一个节点的子节点分为两类：子节点最多的子节点是重子节点，其他的都是轻子节点。

同样的，任意一点连向重子节点的边是重边，其他的是轻边。

所谓重链，就是一条均由重边连成的链。

在进行深搜的时候，我们可以优先搜向重子节点，这样 DFN 序会有非常好的性质。

树链剖分

我们将树上一个节点的子节点分为两类：子节点最多的子节点是重子节点，其他的都是轻子节点。

同样的，任意一点连向重子节点的边是重边，其他的是轻边。

所谓重链，就是一条均由重边连成的链。

在进行深搜的时候，我们可以优先搜向重子节点，这样 DFN 序会有非常好的性质。

即：

同一棵子树内的点在 DFN 序上连续
同一条重链上的点在 DFN 序上连续

这样我们就可以方便地使用连续的区间来维护子树信息和重链信息。

树链剖分

结合这张经典图，可以更方便地理解树链剖分的性质。

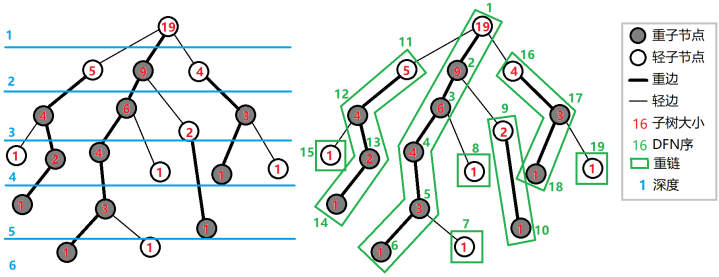


图 1:

树链剖分

另外，利用树链剖分，我们也可以干一些其他操作，比如求两个点 LCA。具体而言，可以这样做：

1. 判断两个点是否在同一条重链上（链顶点是否相同），如果是，那么找深度浅的为 LCA。
2. 深度深的点向父节点跳
3. 不断重复 $1 \sim 2$，直到找到 LCA。

树链剖分求 LCA 的常数较小，所以很有实用意义。

树上随机游走

考虑这样一个问题：

在一棵有根树上，一只蚂蚁初始在根节点处。每一个时刻，它都会向任意方向等概率地移动一条边，问蚂蚁第一次到每一个节点的期望移动次数。

这里我们为了简化问题，暂时不考虑边权。

规定记号如下：

u, v 表示节点
fa_u 表示 u 号节点的父亲节点
son_u 表示 u 号节点的子节点集合
bro_u 表示 u 号节点的兄弟节点集合
d_u 表示 u 号节点的度数

树上随机游走

我们现在求两个函数：

$f(u)$ 为从 u 走向它的父节点的期望次数

$g(u)$ 为从 u 的父节点开始，走到 u 的期望次数

以下列举思路：

$f(u)$：显然地，一个节点走到父节点的期望等于直接走到父节点的期望加上走到子节点再走回去的期望。

$g(u)$：显然地，一个节点被父节点走到的期望等于从父节点直接走来的期望加上绕进兄弟节点再回来的期望加上父亲节点走向祖先节点再走回来的期望。

树上随机游走

$$
\begin{aligned}
f(u) &= \frac{1+\sum_{v\in son_u}(1+f(v)+f(u))}{d_u} \\
&= \frac{1+(d_u-1)(1+f(u))+\sum_{v\in son_u} f(v)}{d_u} \\
d_u f(u) &= d_u + (d_u-1)f(u) + \sum_{v\in son_u} f(v) \\
f(u) &= d_u + \sum_{v\in son_u} f(v)
\end{aligned}
$$

树上随机游走

$$
\begin{aligned}
g(u) &= \frac{1+(1+g(fa_u)+g(u))+\sum_{v\in bro_u}(1+f(v)+g(u))}{d_{fa_u}} \\
&= \frac{d_{fa_u}+(d_{fa_u}-1)g(u)+g(fa_u)+\sum_{v\in bro_u}f(v)}{d_{fa_u}}
\end{aligned}
$$

$$
d_{fa_u}g(u) = d_{fa_u}+(d_{fa_u}-1)g(u)+g(fa_u)+\sum_{v\in bro_u}f(v)
$$

$$
\begin{aligned}
g(u) &= d_{fa_u}+g(fa_u)+\sum_{v\in bro_u}f(v) \\
&= d_{fa_u}+g(fa_u)+f(fa_u)-d_{fa_u}-f(u) \\
&= g(fa_u)+f(fa_u)-f(u)
\end{aligned}
$$

这是作者第一次写原创课件，非常不会，请多担待。

受制于水平原因，很多东西都没有证明，可以自行去 OI-wiki 补一下。

这篇 PDF 不够详细，一些细节还是要靠自己理解和学习。

谢谢