《算法竞赛进阶指南》第 6 版 · 勘误与说明

自第 1 版出版以来，本书受到了广大读者的热情支持，同时也收到了不少有价值的反馈。为了使这本读物能够更好地帮助到广大读者，我们决定在每次批量印刷前，都争取对上一版进行修订，目前是第 6 版。注：版权页标注的是第 1 版第 6 次印刷，但实际上每次印刷都有或大或小的修订，为同之前的版本进行区分，我们称之为第 6 版。

【第 431-432 页】【0x68 二分图的匹配】【KM 算法】

书上现有的算法**时间复杂度应为**$\mathbf{O}(N^2M)$，不够优秀，**更新**$\mathbf{O}(n^3)$**的 KM 算法**：

可以进一步优化，把时间复杂度降低到 $O(N^3)$。注意到每次修改顶标后，相等子图只会扩大，原来相等子图中的边并不会消失。这启发我们每次 DFS 中的很多访问都是冗余的——已经遍历过的交错树没有必要再次遍历。实际上我们只需要从 $A_i + B_j - w(i,j)$ 最小的边，也就是新加入相等子图的边开始继续向下搜索，在已有的交错树基础上继续扩展，而没有必要每次都从交错树的根开始。具体实现中，我们可以记录数组 $upd[j]$ 表示每个右部点 j 对应的最小 Δ，当一个点无法在当前的相等子图中找到匹配时，我们更新顶标，然后直接从最小边开始下一次 DFS。最终匹配成功后，因为 DFS 不一定从根开始，无法利用递归的回溯来更新增广路，我们还需要记录 $last$ 数组表示每一个右部点 j 在交错树中的上一个右部点，沿着 $last$ 倒推回去即可找出增广路。细节请参阅参考程序中的注释。另外，本程序还有非递归的版本。为了方便读者理解，这里给出递归版，非递归版可以参阅 GitHub 上的代码。

```
bool dfs(int x, int fa) {
    va[x] = 1;
    for (int y = 1; y <= n; y++)
        if (!vb[y])
            if (la[x] + lb[y] - w[x][y] == 0) { // 相等子图
                vb[y] = 1; last[y] = fa;
                if (!match[y] || dfs(match[y], y)) {
                    match[y] = x;
                    return true;
                }
            }
            else if (upd[y] > la[x] + lb[y] - w[x][y]) {
                upd[y] = la[x] + lb[y] - w[x][y];
                last[y] = fa;
            }
    return false;
}

long long KM() {
    for (int i = 1; i <= n; i++) {
        la[i] = -(1 << 30); lb[i] = 0;
```

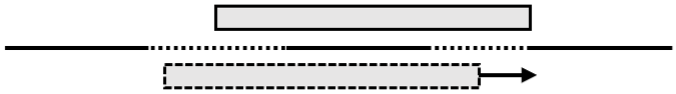
```
            for (int j = 1; j <= n; j++)
                la[i] = max(la[i], 1ll * w[i][j]);
        }
        for (int i = 1; i <= n; i++) {
            memset(va, 0, sizeof(va));
            memset(vb, 0, sizeof(vb));
            memset(last, 0, sizeof(last));
            memset(upd, 0x7f, sizeof(upd));
            // 从右部点 st 匹配的左部点开始 dfs，一开始假设有一条 0-i 的匹配
            int st = 0; match[0] = i;
            while (match[st]) { // 当到达一个非匹配点 st 时停止
                long long delta = 1ll << 60;
                if (dfs(match[st], st)) break;
                for (int j = 1; j <= n; j++)
                    if (!vb[j] && upd[j] < delta) {
                        delta = upd[j];
                        st = j; // 下一次直接从最小边开始 DFS
                    }
                for (int j = 1; j <= n; j++) { // 修改顶标
                    if (va[j]) la[j] -= delta;
                    if (vb[j]) lb[j] += delta; else upd[j] -= delta;
                }
                vb[st] = 1;
            }
            while (st) { // 倒推更新增广路
                match[st] = match[last[st]];
                st = last[st];
            }
        }
        long long ans = 0;
        for (int i = 1; i <= n; i++) ans += w[match[i]][i];
        return ans;
    }
```

【第 419 页】【0x67 Tarjan 算法与有向图连通性】【例题 PKU ACM Team's Excursion】

直接枚举切点如果写得不好，很可能会在一些情况下出现错误。更加标准的一个解法是：

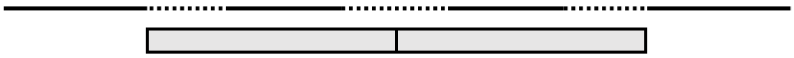
一条最短路其实就是一条“链”，我们要用两个区间覆盖链上两段长度不超过 q 的部分，使未被覆盖的桥的长度之和最小。

如果只有一个区间，那么是非常好做的。当下车位置在“边”的中间而不在“点”上时，把乘车区间向右平移到在“点”上下车，对答案没有影响，如下图所示（虚线表示桥）。因此，可以用双指针（又称尺取法）扫描一遍这条最短路，求出 $ds[i]$，表示从 S 到最短路上的第 i 个节点只搭一次车的最小危险程度。请读者自己写出详细过程。



类似地，还可以倒着扫描一遍，求出 $dt[i]$，表示从最短路上的第 i 个节点到 T，只搭一次车的最小危险程度。最后，我们枚举“切点” i，用 $ds[i]+dt[i]$ 更新答案。

此时我们还遗漏了一种情况——当两个乘车区间相连成为一个长度为 $2q$ 的区间时，它可能无法拆分为两个下（上）车位置在“点”上的乘车区间，如下图所示。这也不难处理，我们只需要用一个长度为 $2q$ 的区间再扫描一次，把答案更新即可。



【第 423 页】【0x67 Tarjan 算法与有向图连通性】【例题 Priest John's Busiest Day】

<u>更新“第二种构造方法”的算法描述和程序片段，更加容易理解：</u>

第二种构造方法在第一种构造方法的基础上，进一步利用了本节给出的“Tarjan 求 SCC 的程序”对 SCC 编号的特殊性质，使得构造过程十分简洁。注意到 Tarjan 算法的本质是一次 DFS，它在回溯时会先取出有向图“底部”的 SCC 进行标记。故 Tarjan 算法得到的 SCC 编号本身就已经满足缩点后的有向无环图中“自底向上”的顺序——序列 $1,2,\dots,cnt$ 就是缩点后的反图的拓扑序，无需进行拓扑排序。

因此，直接比较节点所在的 SCC 的编号大小，即可确定对应变量的赋值 $val[i]$。$c[i]$ 和 $c[n+i]$ 哪个小，就代表哪个会在拓扑序列中先被遍历到，从而决定该 SCC 的变量取值。

```
for (int i = 1; i <= 2 * n; i++)
    if (!dfn[i]) tarjan(i);
for (int i = 1; i <= n; i++)
    if (c[i] == c[n + i]) { puts("NO"); return 0; }
puts("YES");
for (int i = 1; i <= n; i++) val[i] = c[i] > c[n + i];
```

若 $val[i]=0$，则变量 X_i 应赋值为 $X_{i,0}$。若 $val[i]=1$，则变量 X_i 应赋值为 $X_{i,1}$。本书配套光盘提供了该方法的完整参考程序。

【第 448 页】【0x6A 网络流初步】【最小割】

<u>补充最大流最小割定理的证明：</u>

证明：

设在残量网络中，源点 S 可达的点集为 A，不可达的点集为 B，显然 $A\cup B=V$。根据定义，最大流 = 从 S 流出的流量之和 = 从 A 流出的流量之和 – 流入 A 的流量之和。

考虑从 A 流出的流量。对于原图中的任意边 $(x,y)\in E,\ x\in A,\ y\in B$，这些边**一定满流**。若不然，则说明某条边剩余容量大于零，在残量网络中，此时 S 到达 x 后可以沿着这条边到 y，与 $y\in B$ 矛盾。

考虑流入 A 的流量。对于原图中的任意边 $(u,v)\in E,\ u\in B,\ v\in A$，这些边流量**一定为零**。

若不然，说明有流经过某条边，则它对应的反向边 (v,u) 具有大于零的剩余容量。在残量网络中，S 到达 v 后可以沿着这条反向边到 u，与 $u \in B$ 矛盾。

综上所述，我们构造的割集，即原图中 A,B 之间边的容量之和，等于从 A 流出的流量之和，等于最大流。

证毕。