这里讲一下如何把预流推进从5.2s优化到300ms。

实现了一发裸的优先队列维护的预流推进，跑了5.2s。打开提交记录排行榜，榜一@tagyx 跑得飞快，只有150ms，比我快了三十多倍。于是点开他的[提交记录](https://www.luogu.com.cn/record/40938924)学习了一下，成功优化到300ms。（感谢这位大佬加入代码公开计划）

优化主要有四部分：

## 1. bfs&globalRelabel

这一点其他题解提到了。我们发现，如果在初始化时把源点之外的所有结点高度设为0，是浪费时间的。我们可以在初始化时通过一次从汇点t的bfs，预先设定好结点的高度。这并不会影响算法的正确性，原来可以到达汇点t的路径，重设高度后仍然可以到达。

然后我们又发现，这个bfs不仅可以在初始化时进行，也可以在算法运行时（对残余网络）进行，仍然不改变正确性。由此我们又有了一个启发式的想法：每过一段时间，就从汇点t进行一次bfs，对全局重新设定高度，也就是globalRelabel。

那么这个gloablRelabel的频率如何设定呢？bfs的复杂度是*O*(*m*)的，而HLPP算法的复杂度是*O*(*n*2*m*​)，因此程序进行*O*(*n*2 /√*m*​)次globalRelabel不改变复杂度。不妨考虑稠密图*m*=*O*(*n*2)，则globalRelabel可以进行*O*(*n*)次。

算法导论中有证明，原算法Relabel操作（重新设定高度）的次数不超过*O*(*n*2)。因此，我们每*O*(*n*)次Relabel就进行一次globalRelabel可以满足复杂度。考虑到算法的常数，可以每4*n*次或者更多次Relabel进行一次globalRelabel。注意不仅要重新设置高度，很多相关数组也要memset重新搞（所以有一定常数）。

## 2. 不维护高度超过n的结点

HLPP算法的结果是和谐对称的，算法结束时，源点的溢出流为-ans，汇点的溢出流为ans，其他结点均无溢出。

但是我们发现，如果某一结点的高度超过*n*，那么它的溢出流将再也没有推送到汇点*t*的机会。因为溢出流只能推送到高度恰好比它小的结点，也就是说，对于高度为*hx*​的结点，它的溢出流需要沿着一条高度分别为*hx*​,*hx*​−1,...,0的结点构成的路径推送到汇点*t*。如果*hx*​≥*n*，那么就不可能存在这条路径了，因为没有这么多结点。

那么，高度超过*n*的结点，它的溢出流只能以某种方式回到源点*s*。但我们只要关心汇点*t*的溢出流即可。因此，当一个结点的高度超过*n*，我们根本就不需要再维护它，再去对它push/relabel——反正总有一天它的溢出流会回到s的。而且，如果同时应用globalRelabel优化，这些结点也不用参加globalRelabel，相当于被踢出了算法。

这一步优化应该非常重要，减少了大量的工作量。代价就是算法终止时，会有一些结点的溢出流不为零，它们是本该被推送回*s*却没有推送的溢出流。同样，我们也不能用s处的溢出流计算答案，而只能用*t*的溢出流。

## 3. gap优化

当不存在高度为*h*0​的结点时，所有高度大于*h*0​的结点，它们的溢出流都将无法推送到汇点，因此只要把它们的高度设为*n*，不再维护即可。

## 4. 当前弧优化

记录当前弧，在进行多次不饱和推送时有效。这个就不多说了，左转dinic。

## 5. 众所周知四大天王有五个（舍弃优先队列）

用链表维护溢出结点。比如，overList[h]表示高度为h的所有溢出结点的链表的头指针，大概就是这样。然后从maxHeight开始遍历overList数组即可。

这一点我不能确定是不是优化。虽说优先队列维护高度最大的结点有一个log，但如果改用链表维护也不一定是O(1)的，因为有一些overList可能为空。或许用一些骚操作可以摊还到O(1)？

不过话又说回来，前面那些优化，也只有2是明确的优化，1和3都是有点玄学的。也没关系，我们都是玄学家嘛，跑得快就行（点头）

另附一份代码：

struct HLPP {

/\*

\* set ANS\_TYPE to int or long long

\* nodes numbered by 1, 2, ..., n

\*/

typedef long long ANS\_TYPE; // or int

struct Edge {

int to;

ANS\_TYPE c;

int rev;

Edge(int to, ANS\_TYPE c, int rev): to(to), c(c), rev(rev) {}

};

int n, m, s, t;

int maxh, maxgaph, workcnt;

std::vector<std::vector<Edge>> vec; // graph

std::vector<ANS\_TYPE> ov; // overflow of nodes

std::vector<int> h; // height of nodes

std::vector<int> cur; // current arc

std::vector<int> ovList, ovNxt;

std::vector<int> gap, gapPrv, gapNxt;

// n: nodes, m: edges, s: source node, t: sink node

HLPP(int n, int m, int s, int t):

n(n), m(m), s(s), t(t), maxh(0), maxgaph(0), workcnt(0),

vec(n+1), ov(n+1), h(n+1), cur(n+1),

ovList((n+1), -1), ovNxt(n+1, -1),

gap((n+1), -1), gapPrv(n+1, -1), gapNxt(n+1, -1) {}

void addEdge(int u, int v, ANS\_TYPE c) {

vec[u].push\_back(Edge(v, c, vec[v].size()));

vec[v].push\_back(Edge(u, 0, vec[u].size()-1));

}

ANS\_TYPE getMaxFlow() {

globalRelabel();

for(auto &e: vec[s]) if(e.c) { pushFlow(s, e, e.c); maxh = std::max(maxh, h[e.to]); }

for(; maxh >= 0; --maxh) {

while(~ovList[maxh]) {

int x = ovList[maxh];

ovList[maxh] = ovNxt[x];

discharge(x);

if(workcnt > (n<<2)) globalRelabel();

}

}

return ov[t];

}

private:

void discharge(int x) {

int nh = n, sz = vec[x].size();

for(int i = cur[x]; i < sz; ++i) {

auto &e = vec[x][i];

if(e.c > 0) {

if(h[x] == h[e.to]+1) {

pushFlow(x, e, std::min(ov[x], e.c));

if(ov[x] == 0) { cur[x] = i; return ; }

} else nh = std::min(nh, h[e.to]+1);

}

}

for(int i = 0; i < cur[x]; ++i) {

auto &e = vec[x][i];

if(e.c > 0) { nh = std::min(nh, h[e.to]+1); }

}

cur[x] = 0;

++workcnt;

if(~gapNxt[gap[h[x]]]) setHeight(x, nh);

else {

int oldh = h[x];

for(int i = oldh; i <= maxgaph; ++i) {

for(int j = gap[i]; ~j; j = gapNxt[j]) h[j] = n;

gap[i] = -1;

}

maxgaph = oldh-1;

}

}

void globalRelabel() {

workcnt = maxh = maxgaph = 0;

std::fill(h.begin(), h.end(), n); h[t] = 0;

std::fill(gapPrv.begin(), gapPrv.end(), -1);

std::fill(gapNxt.begin(), gapNxt.end(), -1);

std::fill(gap.begin(), gap.end(), -1);

std::fill(ovList.begin(), ovList.end(), -1);

std::fill(ovNxt.begin(), ovNxt.end(), -1);

std::fill(cur.begin(), cur.end(), 0);

std::queue<int> que; que.push(t);

int x;

while(!que.empty()) {

x = que.front(); que.pop();

for(auto &e: vec[x]) {

if(h[e.to] == n && e.to != s && vec[e.to][e.rev].c > 0) {

setHeight(e.to, h[x]+1);

que.push(e.to);

}

}

}

}

void setHeight(int x, int newh) {

if(~gapPrv[x]) {

if(gapPrv[x] == x) {

gapPrv[gapNxt[x]] = gapNxt[x];

gap[h[x]] = gapNxt[x];

} else {

gapNxt[gapPrv[x]] = gapNxt[x];

if(~gapNxt[x]) gapPrv[gapNxt[x]] = gapPrv[x];

}

}

if((h[x] = newh) >= n) return ; // ignore the case of h >= n

maxgaph = std::max(maxgaph, h[x]);

if(ov[x] > 0) { maxh = std::max(maxh, h[x]); ovNxt[x] = ovList[h[x]]; ovList[h[x]] = x; }

if(~(gapNxt[x] = gap[h[x]])) gapPrv[gapNxt[x]] = x;

gap[h[x]] = gapPrv[x] = x;

}

void pushFlow(int from, Edge &e, ANS\_TYPE flow) {

if(!ov[e.to] && e.to != t) {

ovNxt[e.to] = ovList[h[e.to]];

ovList[h[e.to]] = e.to;

}

e.c -= flow;

vec[e.to][e.rev].c += flow;

ov[from] -= flow;

ov[e.to] += flow;

}

};