对网络最大流算法优化与问题归约的研究

陈其轩，毛骏奇，王启帆

摘　要

求解**网络最大流**网络最大流是重要的组合优化问题，也是图论领域的研究热点。我们首先给出最大流问题相关的符号与定义，随后将前人提出的最大流算法与优化分为两类进行归纳总结，并对 HLPP 算法提出了一种改进方法，最后对算法进行性能测试与评价推广，并介绍了数种能够归约为最大流问题的问题类型。

在最大流算法研究部分，我们首先根据算法流程特征，将最大流算法分为了增广路算法与预流推进算法两大类。在增广路算法部分，我们对以 EK 算法，Dinic 算法为代表的传统增广路算法进行了介绍，并总结了其优化方法。随后，我们介绍了在 Dinic 基础上改进产生的 MPM 算法与 ISAP 算法，并总结其优化方法。在预流推进算法部分，我们首先总结了预流推进类算法的特征与通用模板，接着重点介绍了在学术界被广泛认为效率最高的**最高标号预流推进算法** (HLPP 算法)，介绍了两种常见的优化方法。在预流推进算法部分的最后，我们讨论并深入分析了 HLPP 算法效率提升的难点，即“乒乓效应”，介绍、复现了能够有效避免“乒乓效应”的 **Budget 算法**（原论文未给出源码），并使用测试数据进行试验比较上述算法的性能。

在算法研究的末尾，我们通过多篇相关文献总结了近年网络流算法的进展，并从论文了解到近似线性时间复杂度的网络流算法的存在性，并对网络流算法未来可能的发展方向进行了展望。

在问题归约部分，我们整理总结了 4 种可以归约为最大流问题的问题类别，分别为：二分图最大匹配问题、最小费用最大流问题、二分图最大全匹配问题与最小割问题，并对各类问题归约转换的正确性进行证明。

在工作中，包括主流算法研究与问题规约中的所有内容均使用 C++ 程序复现，并使用测试数据进行试验。最后，我们将程序代码放至附录部分。

关键词: 网络最大流; *dinic* 算法; 最高标号预流推进算法; *budget* 算法

I 引入

1.1 最大流问题

现有一由点集 $V = \{v_1, v_2...v_n | n < \infty\}$ 与点集中元素的有序对集合 $E = \{e_1, e_2...e_n | n < \infty\}$ 构成的不包含重边和自环的二元组 $G_e = \{V, E\}$，G_e 称作简单有向图，其中的有序对集 E 称作边集。

我们在 G_e 中加入集合 $C = \{c_{ij} | i, j <= n\}$,其中 c_{ij} 定义为顶点 v_i 与 v_j 之间的容量 (capacity)，得到的 $G = \{V, E, C\}$ 称为容量网络。此外，严格定义的容量网络还需满足 G 中出度为 0 的点 (汇点) 与入度为 0 的点 (源点) 均有且仅有一个。

建立容量网络边集到实数上的映射 $f(u, v)$，其中 $u \in V, v \in V$。若这一实函数满足容量限制、斜对称性与流守恒性，这一函数就称为容量网络的流函数。流函数形式化定义表述如下：

$$f(u,v) = \begin{cases} f(u,v), & (u,v) \in E \\ -f(u,v), & (v,u) \in E \\ 0, & (u,v) \notin E, (v,u) \notin E \end{cases}$$

$$s.t. \begin{cases} f(u,v) < c(u,v) \\ f(u,v) = -f(u,v) \\ \sum_{(u,x)\in E} f(u,x) = \sum_{(x,v)\in E} f(x,v), \quad \forall x \in V - \{s,t\} \end{cases}$$

网络最大流的核心问题是：对于一个容量网络与一对源点、汇点，求解流经该网络的最大流量大小。也即是求解

$$M = max f(s,t)$$

1.2 残量网络

定义 1.1 (剩余容量) 记端点为 u, v 边的容量与流量分别为 $c(u,v)$，$f(u,v)$，那么 $c_f(u,v) = c(u,v) - f(u,v)$ 是该边的剩余容量。

定义 1.2 (残量网络) 容量网络 G 中所有节点与剩余容量为正的边所构成的子图是 G 的残量网络。形式化地，残量网络 G_f 的定义式为：

$$G_f = (V_f = V, E_f = \{(u,v) \in E, c_f(u,v) > 0\})$$

值得注意的是，剩余容量大于 0 的边可能不在原图 G 中，也即是说，残量网络中额外包含了原容量网络中的虚边。

1.3 增广路

定义 1.3 (增广路) 增广路是 G 中从源点至汇点中任意一边剩余容量大于 0 的路径。

相对的，由于残量网络包含了所有正容量边构成的边集，故在残量网络中，任意一条从源点到汇点的路都是增广路。

II 增广路算法

传统网络最大流算法的代表是增广路算法。增广路算法往往通过广度优先搜索不断地寻找可行的增广路进行增广，至网络中不再有增广路时停止算法。EK 算法、Dinic 算法、MPM 算法与 ISAP 算法等大多数传统网络流算法都是增广路算法。

算法	优化方法	时间复杂度	算法类型
Edmonds-Karp 动能算法	无	$O(nm^2)$	增广路算法
Dinic 算法	多路增广与当前弧优化	$O(n^2m)$	增广路算法
MPM 算法	基于堆的优先队列	$O(n^2\log n)$	增广路算法
MPM 算法	BFS 增广路搜索	$O(n^3)$	增广路算法
ISAP 算法	GAP 优化与当前弧优化	$O(n^2m)$	增广路算法
HLPP 算法	GAP 优化与 BFS 优化	$O(n^2\sqrt{m})$	预流推进算法
Budget 算法	无	$\Omega(nm)$	预流推进 + 增广路算法

2.1 Edmonds-Karp 动能算法 (EK 算法)

Edmonds-Karp 动能算法是经典的增广路算法，其时间复杂度为 $O(nm^2)$。

2.1.1 Edmonds-Karp 动能算法流程

具体而言，EK 算法的流程如下。

i. 初始化网络 G，即对所有 $(u_i,u_j)\in E$，$c_f(u_i,u_j)=c_{ij}$，$c_f(u_j,u_i)=0$；其中，边 (u_j,u_i) 称为反向边，其用于撤销非最优的局部极优策略；

ii. 通过广度优先算法寻找增广路 w；若 w 不存在，终止算法；

iii. 寻找 $(u_0,v_0)\in w$，使得对任意 $(u,v)\in w, c_f(u_0,v_0)\le c_f(u,v)$；

iv. 对所有的 $(u,v)\in w$，$c_f(u,v)\leftarrow c_f(u,v)-c_f(u_0,v_0)$，$c_f(v,u)\leftarrow c_f(v,u)+c_f(u_0,v_0)$，同时有 $M\leftarrow M+c_f(u_0,v_0)$；从第二步重复所有步骤。

可以将以上的流程表示为伪代码如下。

2.1.2 Edmonds-Karp 动能算法时间复杂度分析

我们首先证明 EK 算法的每次增广使得所有顶点 $u\in(V-\{s,t\})$ 到 s 的最短距离 $d(u)$ 增加。我们可以采用反证法证明这一观点：

我们记增广后的距离函数为 d'，假设网络中存在一点 $u\in(V-\{s,t\})$，使得 $d'(u)<d(u)$。设 u 的前驱节点为 v，故有

$$d(v)=d(u)-1, d'(v)\ge d(v)$$

这样就有 $(v,u)\notin V$，因为若有 $(v,u)\in V$，就一定有

$$d(u)\le d(v)+1\le d'(v)+1=d'(u)$$

这与假设相矛盾，故 EK 算法的每次增广使得所有顶点 $u\in(V-\{s,t\})$ 到 s 的最短距离 $d(u)$ 增加。

我们再引入关键边概念：

Algorithm 1: Edmonds-Karp Algorithm

Input: edge set E, vertex set V, capacity of each edge C

Output: maximum flow M

for edge (u_i, v_i) in E, **do**
  $c_f(u_i, u_j) \leftarrow c_{ij}$;
  $c_f(u_j, u_i) \leftarrow 0$;
end
Create a queue q; Then append s to it;
while a route from s to t can be found by BFS **do**
  Find the minimun capacity value c_{min} among each vertex in the route;
  for Each vertex (u, v) in the route, **do**
    $c_f(u, v) \leftarrow c_f(u, v) - c_f min$;
    $c_f(v, u) \leftarrow c_f(v, u) + c_f min$;
  end
  $M \leftarrow M + c_{min}$;
end

定义 2.1 (关键边) 若增广路 P 上一边 e 的剩余容量等于增广路的剩余容量，则 e 称为 P 的关键边。

接下来，我们使用引理证明每条边最多作为关键边 $n/2-1$ 次：

对于关键边 (u, v)，由于 (u, v) 在最短路上，有

$$d(v) = d(u) + 1$$

而增广后，(u, v) 将会从网络中消失，重新出现的条件是 (v,u) 出现在增广路上，那么则有

$$d'(u) = d'(v) + 1$$

由引理我们知道：

$$d'(v) \geq d(v)$$

故有

$$d'(u) \geq d(v) + 1 = d(u) + 2$$

所以每次出现至少会使得最短距离 +2，而其距离最大为 n-2，故每条边最多做关键边 $n/2-1$ 次，增广的时间复杂度就为 $O(nm)$。若采用 BFS 进行增广，复杂度就为 $O(nm^2)$。

2.2 Dinic 算法

2.2.1 Dinic 算法流程

Dinic 算法在 EK 算法的基础上引入基于广度优先搜索的分层操作，并利用分层操作和深度优先搜索寻找最短增广路。可以证明，Dinic 算法的时间复杂度是 $O(n^2m)$，在稠密图中显著优于 EK 算法 ($O(nm^2)$)。

具体而言，Dinic 算法的流程如下。

1. 初始化网络 G，即对所有 $(u_i, u_j) \in E$，$c_f(u_i, u_j) = c_{ij}$，$c_f(u_j, u_i) = 0$；
2. 从源点 s 开始通过广度优先搜索为所有结点定义深度 $d(u)$，其中边 (u, v) 可以被经过，当且仅当其剩余容量大于零；u 的深度代表从 s 到 u 的最短路径的长度，且 $d(s) = 0$；若 $d(t)$ 不存在，终止算法；
3. 通过深度优先搜索寻找增广路 w；其中，边 (u, v) 可以成为增广路的一部分，当且仅当 $d(u) + 1 = d(v)$；
4. 寻找 $(u_0, v_0) \in w$，使得对任意 $(u, v) \in w, c_f(u_0, v_0) \leq c_f(u, v)$；
5. 对所有的 $(u, v) \in w$，$c_f(u, v) \leftarrow c_f(u, v) - c_f(u_0, v_0)$，$c_f(v, u) \leftarrow c_f(v, u) + c_f(u_0, v_0)$，同时有 $M \leftarrow M + c_f(u_0, v_0)$；从第二步重复所有步骤。

2.2.2 Dinic 算法优化

Dinic 算法存在两种重要优化。此二种优化可以显著提升 Dinic 算法的效率，因而在实践中常采取此二种优化。

多路增广

利用深度优先搜索确认一增广路 w 后，可能存在 w 的一子序列 $w_0 = \{e_i\}, i = 1, 2, \ldots, k_0 < k$ 满足 $\forall e_i \in w_0, c_f(e_i) > 0$，此时可以继续深度优先搜索寻找增广路，直到未被确认的增广路不再存在。此优化允许 Dinic 算法在同一次深度优先搜索中寻找多条增广路，从而提升算法效率。

当前弧优化

在同一轮深度优先搜索过程中，对结点 u 的第 i 条边 (u, v_i) 进行搜索，当且仅当所有 $(u, v_j), j = 1, 2, \ldots, i-1$ 已经被搜索完毕，后者等价于上述 (u, v_j) 不再可能成为新增广路的一部分。此时可以记录节点 u 正在被搜索的边的下标 i，当再次访问 u 时，可以直接从 (u, v_i) 开始遍历搜索，从而提升效率。

2.2.3 Dinic 算法时间复杂度分析

正如前文所述，Dinic 算法的时间复杂度是 $O(n^2m)$。下方给出证明。

首先考虑单轮增广的时间复杂度。在应用了当前弧优化的情况下，对任意节点，当前弧最多变化 m 次，因此单轮增广的最坏时间复杂度为 $O(nm)$。

现在证明为得到最大流，最多需要进行 $n-1$ 轮增广。

Dinic 算法的任意增广路 w 都满足 $\forall (u, v) \in w, d(v) - 1 = d(u)$。同时，可以发现每轮增广后，$d(t)$ 必定增大。

假设一轮增广过后，$d(t)$ 不变。这就说明，存在一条增广路满足 $\forall (u, v) inw, d(v) - 1 = d(u)$，前者正好符合上一轮增广中增广路的条件，因而其应当在上一轮中被增广。因此，一轮增广过后，不可能存在前述的增广路，也即一轮增广过后，$d(t)$ 必定增大。结合 Dinic 的终止条件，可以知道增广过程最多进行 $n-1$ 次。

综上所述，Dinic 算法的最坏时间复杂度为 $O(n^2m)$。

2.3 MPM 算法

MPM 算法在整体结构上与 Dinic 算法类似，但增广的方法不同，在使用 BFS 解法优化后，时间复杂度优于 Dinic 算法，达到 $O(n^3)$。

为介绍 MPM 算法的增广方式，我们需要引入点容量与参考节点概念。

定义 2.2 (点容量) 点 v 的传入残量与传出残量的最小值是点 v 的点容量。形式化地，点容量 $p(v)$ 的定义式为：

$$p_{in}(v) = \sum_{(u,v)\in L} (c(u,v) - f(u,v))$$

$$p_{out}(v) = \sum_{(v,u)\in L} (c(u,v) - f(u,v))$$

$$p(v) = min(p_{in}(v), p_{out}(v))$$

定义 2.3 (参考节点) 节点 r 是参考节点，当且仅当 $p(r) = minp(v)$

较之 Dinic 算法，MPM 算法增广的对象为顶点，而非边。由于 L 是无环图且 L 中任一节点 v_i 节点容量至少为 $p(v_i)$，故对于每一参考节点 v_i，一定可以使使通过其的流量增加 $p(v_i)$。增加的流量 $p(v_i)$ 被分配至一条从 s 出发经过 v_i 到达 t 的有向路径上。

寻找参考节点的增广路可以有多种方法，使用基于堆的优先队列时间复杂度为 $O(n^2logn)$，而使用 BFS 进行增广路搜寻则可以使时间复杂度达到 $O(n^2)$。

由于参考节点最多不超过 n 个，故增广的迭代最多进行 n^2 次，故 MPM 算法的时间复杂度可以优化至 $O(n^3)$。

可以将 MPM 算法的核心流程表示为伪代码。MPM 算法的核心流程的伪代码请见下文中的 Algorithm 2。

2.4 ISAP 算法

2.4.1 ISAP 算法流程

ISAP 算法，即 Improved Shortest Augmenting Path，在 Dinic 算法的基础上，通过动态地改变节点的深度，将算法中广度优先搜索的次数减少至一次。

具体而言，ISAP 算法的流程如下。

1. 初始化网络 G，即对所有 $(u_i, u_j) \in E$，$c_f(u_i, u_j) = c_{ij}$，$c_f(u_j, u_i) = 0$；
2. 从汇点 t 通过广度优先搜索为所有结点定义深度 $d(u)$，其中 u 的深度代表从 t 到 u 的最短路径的长度，且 $d(t) = 0$；若 $d(s)$ 不存在，终止算法；
3. 通过深度优先搜索寻找增广路 w；其中，边 (u, v) 可以成为增广路的一部分，当且仅当 $d(u) - 1 = d(v)$；
4. 寻找 $(u_0, v_0) \in w$，使得对任意 $(u, v) \in w, c_f(u_0, v_0) \leq c_f(u, v)$；
5. 对所有的 $(u, v) \in w$，$c_f(u, v) \leftarrow c_f(u, v) - c_f(u_0, v_0)$，$c_f(v, u) \leftarrow c_f(v, u) + c_f(u_0, v_0)$，同时有 $M \leftarrow M + c_f(u_0, v_0)$；
6. 对 $w/\{t\}$ 中每个点 u，深度从小到大地判断：若存在相邻结点 v 使得 $c_f(u, v) > 0$ 且 $d(u) - 1 = d(v)$，继续寻找增广路；否则，寻找残量网络 u 的相邻节点集 $\{v\}$ 中点 v_0 使得 $\forall v \in \{v\}, d(v_0) \leq d(v)$，令 $d(u) \leftarrow d(v_0) + 1$；特别地，若 $\{v\} = \varnothing$，令 $d(u) = n$；若 $d(s) \geq n$，终止算法；否则，从第三步开始重复；

Algorithm 2: Malhotra, Pramodh-Kumar and Maheshwari Algorithm

Input: edge set E, vertex set V, capacity of each edge C

Output: maximum flow M

for vertice v_i in V, **do**
  $c_f(u_i, u_j) \leftarrow c_{ij}$;
  $c_f(u_j, u_i) \leftarrow 0$;
end
for vertice v_i in V, **do**
  for vertice pair (u, v) in V, **do**
    $p_{in}(v) += (c(u, v) - f(u, v))$;
    $minp(v) = min(p_{in}(v_i), min_p(v))$;
  end
  for vertice pair (v, u) in V, **do**
    $p_{out}(v) += (c(u, v) - f(u, v))$;
    $minp(v) = min(p_{out}(v_i), min_p(v))$;
  end
end
Create a vector v;
for vertice r_i in V, **do**
  if $p(r_i) == minp(v)$ **then**
    $v.push\_back(r_i)$;
  end
end
while !$v.empty()$ **do**
  Do BFS to find the route from s to t through r_i;
  $ans += p(r_i)$;
  $v.pop(r_i)$;
end

2.4.2 ISAP 算法优化

ISAP 算法也存在两种重要优化。

当前弧优化
当前弧优化对 ISAP 算法是同样成立的。需要注意的是，当一结点 u 的深度改变时，需要将当前弧重置为第一条边。

GAP 优化
在进行广度优先搜索时，记录每个深度下节点的个数 $n_d(i)$。由于增广路 w 中任意边 (u,v) 满足 $d(u)-1=d(v)$，若 $\exists n_d(i)=0$，则图中不再可能存在新增广路。此时可以直接终止算法以提升效率。

III 预流推进算法

预流推进算法是一种使溢出节点尽可能向下一高度推流，进行最大流求解的高效算法，与先前的增广路算法不同，预流推进类算法不寻找增广路，容量、残量网络也不遵循流守恒性。其中的最高标号预流推进算法时间复杂度为**紧确** $O(n^2\sqrt{m})$，是目前效率最高的网络流算法。

在介绍通用预流推进算法前，我们需要引入通用的相关定义：

3.1 符号定义

定义 3.1 (超额流) 若容量网络允许流入某一节点 u 的流超出流出其的流，超过的部分被称为节点 $u(u\in V-\{s,t\},)$ 的超额流。形式化地，点 u 的超额流 $e(u)$ 的定义式为：

$$e(u)=\sum_{(x,u)\in E} f(x,u)-\sum_{(u,y)\in E} f(u,y)$$

当 $e(u)>0$ 时，我们称节点 u **溢出**。在预流推进算法中，节点的溢出与超额流的出现是被允许的。

定义 3.2 (高度函数) 与传统网络流算法中的高度函数概念相较更为宽泛，从点集映射至实数集上，满足一定条件的函数称为高度函数。形式化地，残量网络 $G_f=(V_f,E_f)$ 的高度函数 h 需满足如下条件：

$$h(s)=|V|,h(t)=0$$

$$\forall(u,v)\in E_f,h(u)\le h(v)+1$$

从高度函数的定义中，我们可以得到一个重要结论，即预流推进算法的推送执行只发生在 $h(u)=h(v)+1$ 的边上。

3.2 操作定义

定义 3.3 (推流 (Push)) 将溢出节点 u 的超额流推至下一节点 v 的操作为推流操作（若 v 溢出，我们需要对 v 进行相应的标记，在 *HPLL* 中为放入堆中）。形式化地，推流的适用条件为：对 u，$\exists v((u,v)\in E_f,c(u,v)-f(u,v)>0,h(u)=h(v)+1)$。

若推流后 $f(u,v)$ 恰好为 $c(u,v)$，那么称 (u,v) 此时**满流**，将其从残量网络中删去。

定义 3.4 (重贴标签 (Relabel)) 为解决溢出节点无法推流（无高度函数较低的节点）问题而将溢出节点高度提高的操作是重贴标签操作。形式化地，重贴标签操作的适用条件可表述如下：

$$\forall(u,v) \in E_f, h(u) < h(v),$$

$$where : e(u) = \sum_{(x,u)\in E} f(x,u) - \sum_{(u,y)\in E} f(u,y) > 0$$

更新具体操作为：

$$h(u) = min_{(u,v)\in E_f} h(v) + 1$$

定义 3.5 (初始化) 经典的预流推进算法初始化即是将源点高度函数置为 $|V|$（即点的数量），除源点以外节点的高度函数置零，并计算每个节点的超额流。形式化地，初始化操作定义如下：

$$\forall(u,v) \in E, f(u,v) = \begin{cases} c(u,v), & u = s \\ 0, & u \neq s \end{cases}$$

$$\forall u \in V, h(u) = \begin{cases} |V|, & u = s \\ 0, & u \neq s \end{cases}$$

$$e(u) = \sum_{(x,u)\in E} f(x,u) - \sum_{(u,y)\in E} f(u,y)$$

3.3 通用预流推进算法

预流推进类算法共用的模板可以概括为：对图中所有满足进行推流与重贴标签操作进行条件的节点，进行这两项操作直到图中不再有这样的节点。其算法伪代码展示如下：

Algorithm 3: General Preflow Push Algorithm

Input: edge set E, vertex set V, capacity of each edge C

Output: maximum flow M

```
1  init();// Initiate with steps given above;
2  do
3      Unmark all vertexs;
4      for vertex v in V do
5          Push if conditions for pushing are satisfied;
6          Relabel if conditions for relabeling are satisfied;
7          if Pushed or Relabeled then
8              Mark v;
9          end
10     end
11 while at least one marked vertex can be found;
12 M ← e(t);
```

3.4 最高标号预流推进 (HLPP)

最高标号预流推进算法算法被广泛认为是预流推进算法中最优秀的算法。我们通过查阅论文，总结了经典 HLPP 算法的流程。具体而言，经典 HLPP 算法的流程如下。

1. 按照上述给出的定义进行初始化操作;
2. 将网络中发生溢出的节点放入以高度为键值的大根堆中;
3. 选择高度最高的节点 u_i，对其所有可推送的边进行推送，将推送后发生溢出的节点加入大根堆;
4. 如果 u_i 仍然溢出，那么重贴标签后返回步骤 3;
5. 如果 u_i 不溢出，那么 u_i 移除大根堆;
6. 堆为空时，算法结束;

其伪代码见4。

3.5 HLPP 常见优化

经典 HLPP 算法的 Relabel 与 Push 操作是较为冗余的，尤其是 Relabel 操作可以通过一些简单优化显著减少。接下来介绍常用的 BFS 优化与 GAP 优化，现今使用的 HLPP 算法往往会用上这些简单优化。

3.5.1 BFS 优化

在 HLPP 算法的实际应用中，改进初始化方式可以使得 Relabel 操作次数可以被大大减少。

具体而言，我们会对初始化高度这一操作进行修改。在初始化进行广度优先搜索时，我们将节点的高度函数初始化为该节点到汇点的距离，特别的，源点的高度函数值仍为点集大小。形式化的，BFS 优化后的 HPLL 初始化定义如下：

$$\forall u \in V, h(u) = \begin{cases} |V|, & u = s \\ d(u,t), & u \neq s \end{cases}$$

3.5.2 GAP 优化

HLPP 算法同样具有 GAP 优化。从经典 HLPP 算法的流程中，我们不难看出，不可向下推送超额流的溢出节点最终会随着 Relabel 的不断进行高度不断升高，直至高度超过源点，将这部分超额流推送至源点中。

故在算法运行的某一时刻，若存在一个高度 h_g，高度函数值为 h_g 的节点个数为 0，则说明出现了 “断层 “，也即 GAP。此时任何高度函数值大于 h_g 的溢出节点不可能对高度低于 h_g 的节点进行推送，我们可以将这些节点的高度 Relabel 为 $n+1$，使其直接将超额流推送回汇点。

一种实现方式是由 [1] 提出的，采用了桶的思想，开 $2 * N - 1$ 个桶用于记录高度至节点的映射，即桶 $B[i]$ 记录了当前高度为 i 的溢出节点。

Algorithm 4: Highest Label Preflow Push Algorithm

Input: edge set E, vertex set V, capacity of each edge C

Output: maximum flow M

for vertice pair (u_i, v_i) in E, **do**
  if $u == s$ **then**
    $f(u, v) \leftarrow c(u, v)$;
  end
  else
    $f(u, v) \leftarrow 0$;
  end
end
for vertice u_i in V, **do**
  if $u == s$ **then**
    $h(u, v) \leftarrow |V|$;
  end
  else
    $h(u, v) \leftarrow 0$;
  end
  $e(u) \leftarrow sum(f(x, u)) - sum(f(u, y))$;
end
Create a max heap s;
for vertice u_i in V, **do**
  if u_i *is overflow* **then**
    $s.insert(u_i)$;
  end
end
while !$s.empty()$ **do**
  let u_i be the highest vertice in s;
  while u_i is overflow **do**
    Push (u_i, v_i) or Relabel u_i;
    if v_i *is overflow* **then**
      $s.insert(v_i)$;
    end
  end
  s.delete(u_i);
end

3.6 乒乓效应

即使使用了常见优化（BFS 优化、GAP 优化），HLPP 仍无法克服被称为“乒乓效应”的预流推进类算法通病。在当今的学术界，有许多算法领域的学者致力于改进 HLPP 使其规避这一效应，以求将 HLPP 提到更高的效率。我们将通过一个简单的例子来描述乒乓效应，而后给出乒乓效应的形式化定义。为此，我们给出如下初始图：

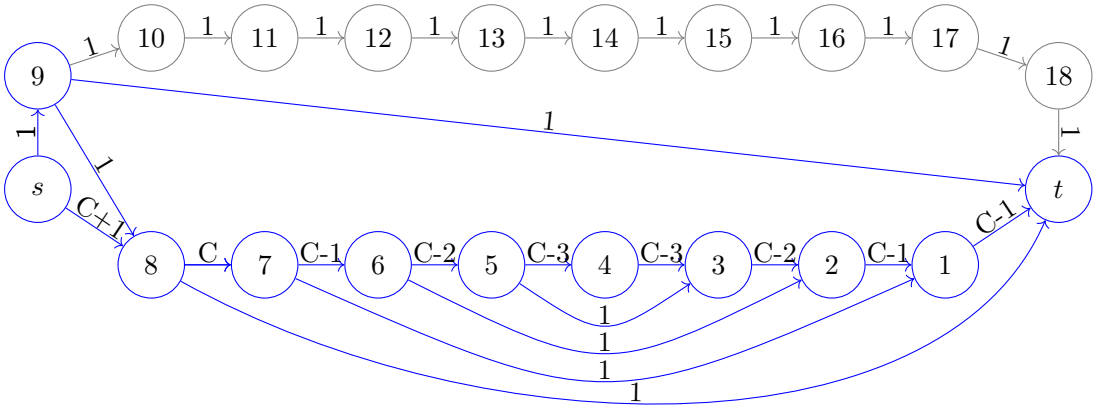

Figure 1. HLPP 算法的初始图

我们在图上运行 HLPP 算法，在经过数次迭代之后，我们得到如下残量网络：

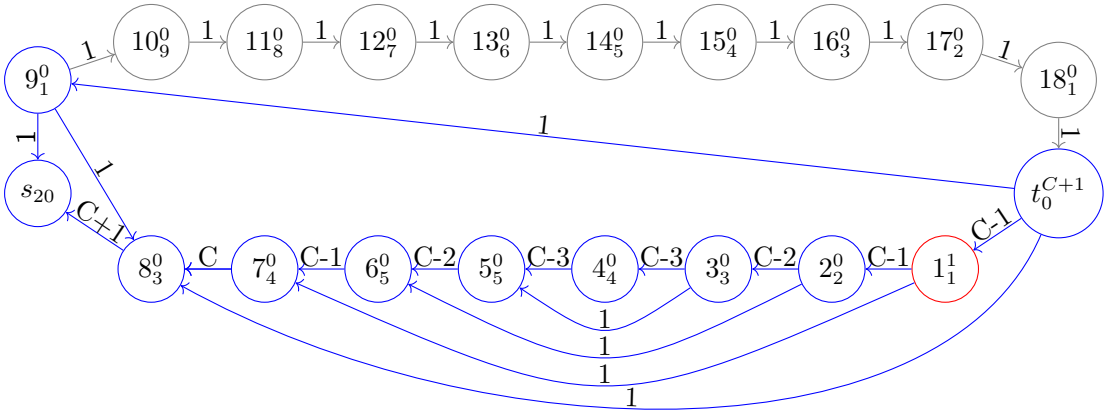

Figure 2. 若干次操作后的残量网络

对于图中的每个节点 u_i，我们都标注出其高度与超额流。具体而言，对于每个节点，我们记录其为 p_d^e，其中 p 为该节点的编号，e 为超额流，d 为其高度。对于每条边，边上的数字代表其剩余容量。

此时，图中具有最高高度函数值的溢出节点为节点 1，在算法继续执行时，节点 1 将被选中，高度函数值变为 3，并向节点 2 推送一单位流量，使节点 2 溢出，节点 1 自身不再溢出。

随后，节点 2 将被算法选中，高度函数值变为 4，并向节点 3 推送一单位流量，使节点 3 溢出，节点 2 自身不再溢出。

随着 HLPP 算法的不断进行，这一单位的流量将被推送至节点 8，各流与高度函数、各边剩余容量如图 3 所示。

由于图中仅有节点 8 是可被选中的，算法将选择节点 8，将其高度函数值变为 5，并向节点 7 推送一单位流量，使节点 7 溢出，节点 8 自身不再溢出。随后，算法又将选择节点 7，并向节点 6

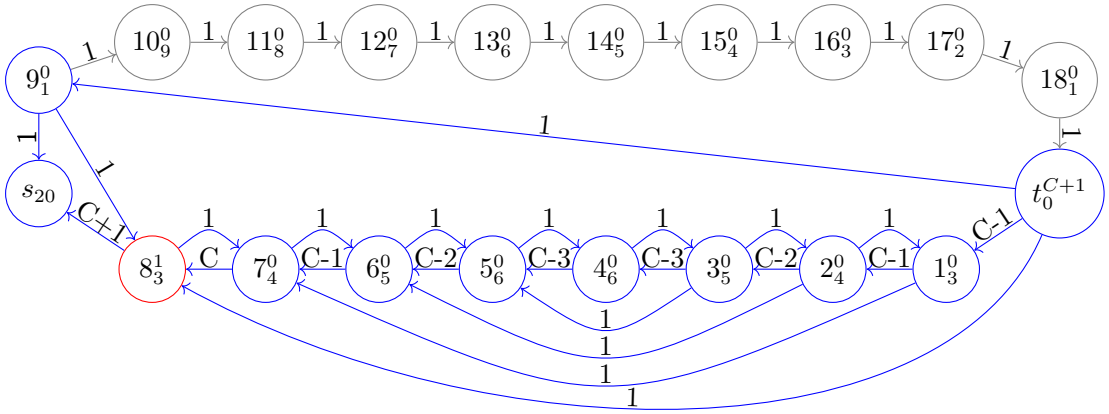

Figure 3. 推流完成之后的残量网络

推送一单位流量，使节点 6 溢出，节点 7 自身不再溢出。算法对路径 $\{6,5,4,3,2,1\}$ 上的每个节点依次进行 Relabel 与 Push 操作。在到达节点 1 后，算法将再次对路径 $\{1,2,3,4,5,6\ 7\ 8\}$ 上的每个节点依次进行 Relabel 与 Push 操作。以上两轮推流的情况可以参照图 4 和图 5。

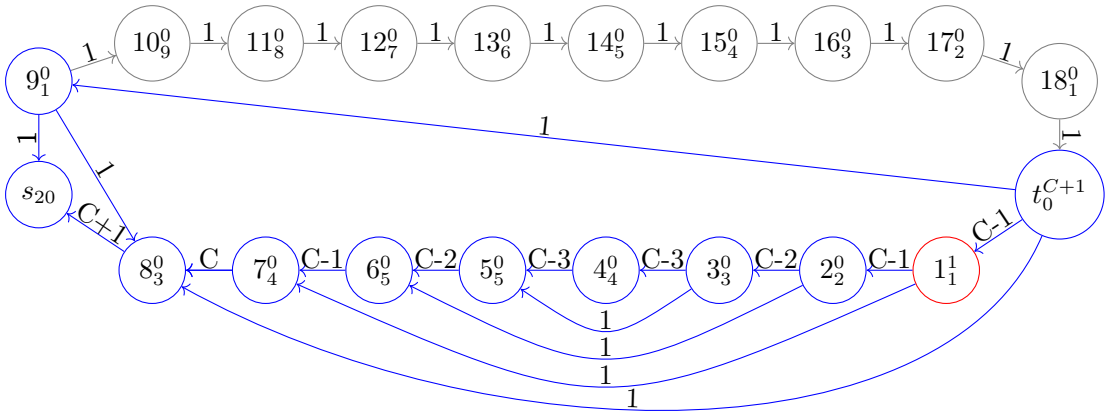

Figure 4. 推流至节点 1 时的残量网络

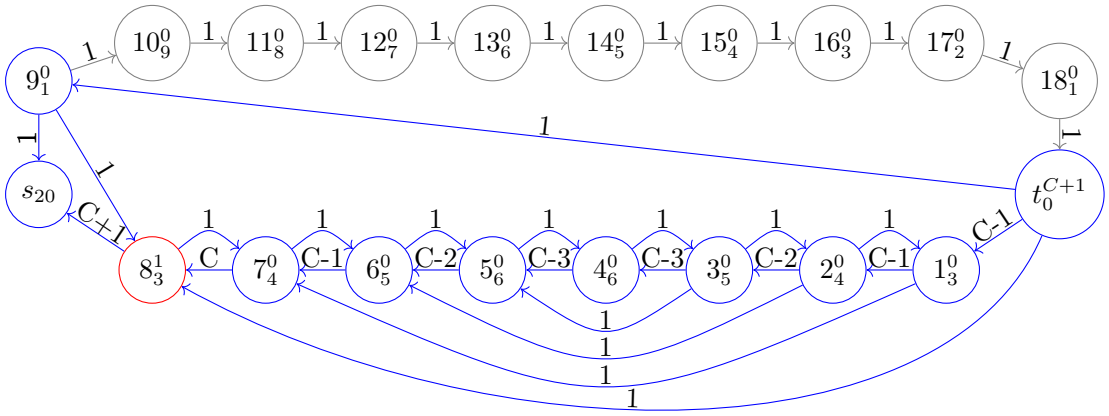

Figure 5. 再次推流至节点 8 时的残量网络

我们不难发现，经过 n+2 次一端至另一端的 Relabel-Push 操作的残量网络与经过 n 次的残量网络的结构完全一致，标签中也仅有高度函数标签与之相比增加了 2。也即是说，HLPP 在运行过

程中多次地交替形成了除高度标签缓慢上升以外完全一致的残量网络。而且在这个过程中，网络并不满足 GAP 优化的使用条件，这样的操作将持续进行直至图中节点高度超过 n，共执行 n 次。这样的现象被称作“乒乓效应”。

一般地，会产生乒乓效应的网络图如图 6 所示。当 HLPP 在该图上运行时，推送会沿路径 $\{1, 2, 3, ..., n/2-2\}$、$\{n/2-2, n/2-3, ..., 2, 1\}$ 反复进行 Relabel-Push 操作 n 次，而每一次沿路径将进行 $n/2-3$ 次推送操作，推送操作是 $\Omega(n^2)$ 的。

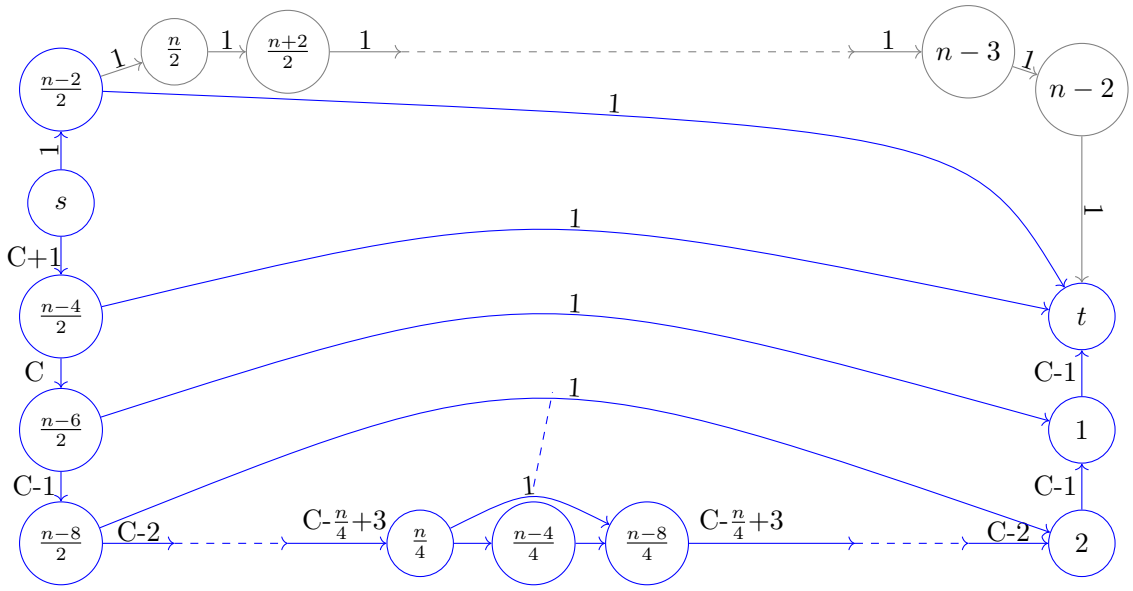

Figure 6. 乒乓效应的一般情形

3.7 Budget 算法

Budget 算法由 [2] 提出，是一种结合了增广路部分特点，可以有效规避乒乓效应的预流推进算法。在介绍乒乓效应时，我们不难发现，HLPP 只会简单地顺着容许弧进行推流，而不考虑后继节点是否能够真正的“消化”推送的流量。一种自然的解决思路是以当前高度最高的溢出节点为起点搜索长度至少为 k 的，并沿着这一路径推送。

为此，我们可以引入以下定义：

定义 3.6 (Budget) *初始化时，对应一个常数 k，节点 u_i 的 Budget 为 $kd(u_i)$。*

当算法选择高度最高的溢出节点 u_i 时，u_i 将会初始化 Budget，并开始尝试构建一条朝向汇点的容许路径。路径并不一定要达到汇点，也可以只在另一节点 u_j 上拓展部分路径，并将 Budget 减去 $d(u_j)$。当 Budget 为负时，向拓展至的节点进行推送操作。

若从节点 u_i 出发找不到这样的路径，也即是说，在 Budget 为正的情况下，无法继续寻找下一个容许弧，此时算法收缩路径，删除终端顶点，并尝试从其他节点拓展路径。

有许多学者在这一思想上提出了具体的实现方法，如 [5] 提出了一种使用动态树形数据结构储存的实现，其开销为 $O(nm \log n^2/m)$，而 [6] 则介绍了一种固定拓展路径长度的算法（根据 [2]，这一做法并不高效）。

我们将 Budget 算法的伪代码展示如下：

Algorithm 5: Budget Algorithm

Input: edge set E, vertex set V, capacity of each edge C

Output: maximum flow M

```
1  init(G);// Initiate with steps given above;
2  while there exists overflow nodes;
3   do
4      i ← argmax{d(i)|i ∈ N, e(i) > 0};
5      budget ← kd(i);
6      i_t ← i;
7      while budget ⩾ 0 and i_t ≠ s and i_t ≠ t;
8       do
9          if there exists an admissible arc (i_t, j) ∈ δ_G^+(i_t);
10          then
11             succ(j) ← NIL;
12             pred(j) ← i_t;
13             succ(i_t) ← j;
14             i_t ← j;
15             budget ← budget − d(j);
16         end
17         else
18             d′ ← min{d(j) + 1|(i, j) ∈ δ_G^+(i_t)};
19             if i_t ≠ i then
20                budget ← budget + d(i_t);
21                i_t ← pred(i_t);
22             end
23             else
24                budget ← kd′;
25             end
26             d(i_t) ← d′;
27             succ(i_t) ← NIL;
28         end
29     end
30     augment(i);
31 end
32 M ← e(t);
```

IV 近似线性复杂度的网络流算法

[3] 给出了一种算法，能够在 $m^{1+o(1)}$ 时间内求解整数流量网络中的最大流与费用流算法。算法通过一系列 $m^{1+o(1)}$ 近似无向最小比率循环进行流的构建，每个循环都使用一种新型动态数据结构进行处理与计算。

4.1 改进的 IPM

IPM（内点法）在最大流的中的应用源于 [4]，文章将内点法与快速拉普拉斯系统求解器相结合，实现了费用流的 $O(m^{1.5}log^2U)$ 算法。IPM 应用于网络流问题的优势在于能够将有向图上的网络流问题减少为更易于处理的无向图上的问题。

论文提出了一种改进的 IPM 算法，下记 ℓ_1IPM 算法（此处的 ℓ_1 即是第一范式）。

Theorem 1 *ℓ_1IPM 算法能够将费用流问题分解为顺序求解无向最小比率循环的 $m^{1+o(1)}$ 个实例，每个实例都达到 $m^{o(1)}$ 近似。*

此外，分解生成的问题实例是“稳定的”，即它们满足：

1) 从当前流到未知的最优流的方向对于每个实例都是一个足够好的解决方案

2) 长度和梯度输入实例的参数仅针对每次迭代的分摊 $m^{o(1)}$ 边发生变化。

经典的 IPM 方法将最小成本流减少到求解 $\widetilde{O}(\sqrt{m})$ 个实例以保证精度，这是一个 ℓ_2 最小化问题。而无向最小比率循环是一个 ℓ_1 最小化问题，它要更为简单，进而可以负担得起子问题中的 $m^{o(1)}$ 近似因子。

4.2 动态延伸数据结构

ℓ_1IPM 算法在每次 IPM 迭代后仅重建低延伸生成树的一部分以适应$\boxed{E}$度的变化，为此需要一种新型的数据结构，该结构在边与顶点较少的图上维护最小比率循环问题实例的递归序列。这些较小的实例给出了更差的近似解，但维护成本更低。我们利用 IPM 的稳定性，将顶点部分嵌入树中。

Theorem 2 *动态延伸数据结构能够对一个未加权的、无向的图 G 进行边缘更新（插入/删除/顶点分裂）。维护一个具有 $O(n)$ 条边的子图 H，以及每个 $e \in H$ 到长度为 $m^{o(1)}$ 的 H 的显式路径嵌入。对于每次边更新，H 中的边更改的摊销数量为 $m^{o(1)}$。此外，嵌入固定边 $e \in H$ 的边集对于所有边 e 都是递减的。*

4.3 伪代码

将论文中提出的近似线性复杂度费用流伪代码见6。

V 问题规约

5.1 二分图最大匹配问题

二分图是节点由两个集合组成，且两个集合内部没有边的图。给定一个二分图，其左部点的个数为 n，右部点的个数为 m，边数为 e，求其最大匹配的边数。

二分图最大匹配问题可以转换成网络流模型。

Algorithm 6: MinCostFlow($G, d, c, u+, u-, f(0), F*$)

Input: graph G, demands d,costs c, upper/lower capacities $u+$,$u-$,initial feasible flow $f(0)$, the optimal flow F^*

Output: maximum flow f^t

$\alpha \leftarrow 1/(1000logmU)$;
$\kappa \leftarrow exp(-O(log7/8mloglogm))$;
$d \leftarrow O(log^{1/8}n)$;
$D^{(HSFC)}$; // Dynamic datastructure;
$T1, T2, ..., Tsfors \leftarrow O(logn)^d$;
$\epsilon \leftarrow \kappa\alpha/(1000s)$;
$r \leftarrow \infty$;
while $\mathrm{c^T} > \mathrm{f(t)-F^*\ (mU)^{-10}}$ **do**
  if t is a multiple of b $\in \lfloor \epsilon c \rfloor$;
  then
    Explicitly compute $f^{(t)} \leftarrow f^{(0)} + \sum_{i\in[s]} f_i, \tilde{f}(t) \leftarrow f(t)$;
    $r \leftarrow c^T\tilde{f}(t) - F^*$;
    $g^{(t)} \leftarrow g(\tilde{f}(t)), l^{(t)} \leftarrow l^{\tilde{f}(t)}$;
    Rebuild $D^{(HSFC)}$ *and update the* T_i;
  end
  $U^{(t)} \leftarrow \bigcup_{i\in[s]} D^{(Ti)}.Detect()$;
  for $e \in U^{(t)}$ **do**
    $Set\tilde{f}_e^{(t)} \leftarrow f_e^{(t)} = f_e^{(0)} + \sum_{i\in[s]}(f_i)_e, l_e^{(t)} \leftarrow l_e^{\tilde{f}(t)}$;
    $g_e^{(t)} \leftarrow 20mc_e/r + \alpha(u_e^+ - \tilde{f}_e^{(t)})^{(-1-\alpha)} - \alpha(\tilde{f}_e^{(t)} - u_e^-)^{(-1-\alpha)}$;
  end
  for $e \notin U^{(t)}$ **do**
    $g_e^{(t)} \leftarrow g_e^{(t-1)}, l_e^{(t)} \leftarrow l_e^{(t-1)}, \tilde{f}_e^{(t)} \leftarrow \tilde{f}_e^{(t-1)}$;
  end
  $D^{(HSFC)}.Update(U^{(t)}, g^{(t)}, l^{(t)}), and\ update\ the\ T_i\ for\ i \in [s]$;
  $(i, \Delta) \leftarrow D^{(HSFC)}.Query(), where\ i \in [s]\ and$;
  $\Delta = (u1, v1) \oplus T_i[v1, u2] \oplus (u2, v2) \oplus ... \oplus (ul, vl) \oplus T_i[v_l, u_1]\ for\ edges\ (u_i, v_i)\ and\ l\ m^{o(1)}$;
  $\Delta \leftarrow \eta\Delta for \eta \leftarrow -\kappa^2\alpha^2/(800hg^{(t)}, \Delta_i)$;
  $f_i \leftarrow f_i + \Delta\ using D^{(Ti)}$;
  $t \leftarrow t + 1$;
  $return\ f^{(t)}$;
end
$M \leftarrow e(t)$;

将源点连上左边所有点，右边所有点连上汇点，容量皆为 1。原来的每条边从左往右连边，容量也皆为 1，最大流即最大匹配。

如果使用 Dinic 算法求该网络的最大流，可在 $O(\sqrt{n}m)$ 求出。

Dinic 算法分成两部分，第一部分用 $O(m)$ 时间 BFS 建立网络流，第二步是 $O(nm)$ 时间 DFS 进行增广。

但因为容量为 1，所以实际时间复杂度为 $O(m)$。

接下来前 $O(\sqrt{n})$ 轮，复杂度为 $O(\sqrt{n}m)$。$O(\sqrt{n})$ 轮以后，每条增广路径长度至少 $\sqrt{n}$，而这样的路径不超过 $\sqrt{n}$，所以此时最多只需要跑 $\sqrt{n}$ 轮，整体复杂度为 $O(\sqrt{n}m)$。

5.2 最小费用最大流

给定一个网络 $G=(V,E)$，每条边除了有容量限制 $c(u,v)$，还有一个单位流量的费用 $w(u,v)$。

当 (u,v) 的流量为 $f(u,v)$ 时，需要花费 $f(u,v)\times w(u,v)$ 的费用。

w 也满足斜对称性，即 $w(u,v)=-w(v,u)$。

则该网络中总花费最小的最大流称为最小费用最大流，即在最大化 $\sum_{(s,v)\in E} f(s,v)$ 的前提下最小化 $\sum_{(u,v)\in E} f(u,v)\times w(u,v)$。

只需将 EK 算法或 Dinic 算法中找增广路的过程，替换为用最短路算法寻找单位费用最小的增广路即可。

5.3 二分图最大权匹配

二分图的最大权匹配是指二分图中边权和最大的匹配。

给定一个二分图，其左部点的个数为 n，右部点的个数为 m，边数为 e，每条边的边权为 w_i。

二分图最大权匹配可以转化为费用流模型。

在图中新增一个源点和一个汇点。

从源点向二分图的每个左部点连一条流量为 1，费用为 0 的边，从二分图的每个右部点向汇点连一条流量为 1，费用为 0 的边。

接下来对于二分图中每一条连接左部点 u 和右部点 v，边权为 w 的边，则连一条从 u 到 v，流量为 1，费用为 w_i 的边。

求这个网络的最大费用最大流即可得到答案。

5.4 最小割问题

对于一个网络流图 $G=(V,E)$，其割的定义为一种点的划分方式：将所有的点划分为 S 和 $T=V-S$ 两个集合，其中源点 $s\in S$，汇点 $t\in T$。

我们的定义割 (S,T) 的容量 $c(S,T)$ 表示所有从 S 到 T 的边的容量之和，即 $c(S,T)=\sum_{u\in S,v\in T} c(u,v)$。当然我们也可以用 $c(s,t)$ 表示 $c(S,T)$。

最小割就是求得一个割 (S,T) 使得割的容量 $c(S,T)$ 最小。

可以证明 $f(s,t)_{\max}=c(s,t)_{\min}$

对于任意一个可行流 $f(s,t)$ 的割 (S,T)，我们可以得到：

$$f(s,t) = S\text{出边的总流量} - S\text{入边的总流量} \leq S\text{出边的总流量} = c(s,t)$$

如果我们求出了最大流 f，那么残余网络中一定不存在 s 到 t 的增广路经，也就是 S 的出边一定是满流，S 的入边一定是零流，于是有：

$$f(s,t) = S\text{出边的总流量} - S\text{入边的总流量} = S\text{出边的总流量} = c(s,t)$$

结合前面的不等式，我们可以知道此时 f 已经达到最大。

如果需要在最小割的前提下最小化割边数量，那么先求出最小割，把没有满流的边容量改成 inf，满流的边容量改成 1，重新跑一遍最小割就可求出最小割边数量；如果没有最小割的前提，直接把所有边的容量设成 1，求最小割即可得到答案。

VI 算法测试

6.1 数据集说明

我们共使用 5 个数据集进行算法性能测试。所使用的数据集均为随机生成，固耗时低于理论时间复杂度。

6.2 运行结果

算法	n	m	备注	通过耗时
Dinic 算法	1200	120000	测试数据 1	531ms
HLPP 算法	1200	120000	测试数据 1	46ms
Budget 算法	1200	120000	测试数据 1	46ms
Dinic 算法	2000	200000	测试数据 2	1468ms
HLPP 算法	2000	200000	测试数据 2	78ms
Budget 算法	2000	200000	测试数据 2	78ms
Dinic 算法	1000	400000	测试数据 3	15656ms
HLPP 算法	1000	400000	测试数据 3	171ms
Budget 算法	1000	400000	测试数据 3	187ms
Dinic 算法	1500	600000	测试数据 4	31281ms
HLPP 算法	1500	600000	测试数据 4	250ms
Budget 算法	1500	600000	测试数据 4	265ms
Dinic 算法	3000	600000	测试数据 5	17187ms
HLPP 算法	3000	600000	测试数据 5	296ms
Budget 算法	3000	600000	测试数据 5	312ms

参考文献

[1] Ravindra K. Ahuja, Murali Kodialam, Ajay K. Mishra, and James B. Orlin. Computational investigations of maximum flow algorithms. *European Journal of Operational Research*, 97(3):509–542, 1997.

[2] R. Cerulli, M. Gentili, and A. Iossa. Efficient preflow push algorithms. *Computers & Operations Research*, 35(8):2694–2708, 2008. Queues in Practice.

[3] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. 2022.

[4] S. I. Daitch and D. A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. *arXiv e-prints*, 2008.

[5] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, oct 1988.

[6] Giuseppe Mazzoni, Stefano Pallottino, and Maria Grazia Scutellà. The maximum flow problem: A max-preflow approach. *European Journal of Operational Research*, 53(3):257–278, 1991.