Efficient Algorithm for Computing All Low *s-t* Edge Connectivities in Directed Graphs

Xiaowei Wu and Chenzi Zhang

The University of Hong Kong
xwwu, czzhang@cs.hku.hk

Abstract. Given a directed graph with n nodes and m edges, the (strong) edge connectivity $\lambda(u, v)$ between two nodes u and v is the minimum number of edges whose deletion makes u and v not strongly connected. The problem of computing the edge connectivities between all pairs of nodes of a directed graph can be done in $O(m^\omega)$ time by Cheung, Lau and Leung (FOCS 2011), where ω is the matrix multiplication factor (≈ 2.373), or in $\tilde{O}(mn^{1.5})$ time using $O(n)$ computations of max-flows by Cheng and Hu (IPCO 1990).
We consider in this paper the "low edge connectivity" problem, which aims at computing the edge connectivities for the pairs of nodes (u, v) such that $\lambda(u, v) \leq k$. While the undirected version of this problem was considered by Hariharan, Kavitha and Panigrahi (SODA 2007), who presented an algorithm with expected running time $\tilde{O}(m+nk^3)$, no algorithm better than computing all-pairs edge connectivities was proposed for directed graphs. We provide an algorithm that computes all low edge connectivities in $O(kmn)$ time, improving the previous best result of $O(\min(m^\omega, mn^{1.5}))$ when $k \leq \sqrt{n}$. Our algorithm also computes a minimum u-v cut for each pair of nodes (u, v) with $\lambda(u, v) \leq k$.

1 Introduction

Given an undirected graph, the edge connectivity between two nodes is the minimum number of edges whose deletion disconnects those two nodes, which by Menger's Theorem [12] is also the maximum number of edge-disjoint paths between them. The definition of edge connectivity can be naturally generalized to directed graphs [13] [1] [6] (it is denoted by "strong edge connectivity" in some literatures): given a digraph $G(V, E)$, the edge connectivity $\lambda(u, v)$ between two nodes $u, v \in V$ is the minimum number of edges whose deletion makes u and v not strongly connected. The edge connectivity of a graph is the minimum edge connectivity between any two nodes in the graph. Computing the edge connectivity is a classic and well-studied problem.

Given two nodes u and v in a digraph, the edge connectivity $\lambda(u, v) = \min\{f(u, v), f(v, u)\}$, where $f(u, v)$ is the max-flow from u to v, if we attach unit capacity to each edge. Given a unit capacity network with m edges and n nodes, Even and Tarjan [5] showed that Dinic's algorithm [4] for computing the s-t max-flow terminates in $O(\min\{m^{\frac{3}{2}}, mn^{\frac{2}{3}}\})$ time. The above algorithm was

the fastest algorithm for computing unit capacity max-flow for almost 40 years until very recently Lee and Sidford [11] proposed an $\tilde{O}(m\sqrt{n})$ time algorithm using a new method to solve LP.

The problem of computing the edge connectivities between all pairs of nodes of a digraph was also considered. Note that the problem can be trivially solved by computing $O(n^2)$ max-flows, which yields a total running time of $\tilde{O}(mn^{2.5})$ by Lee and Sidford [11]. Cheung et al. [3] considered the problem and proposed an $O(m^\omega)$ time randomized algorithm, where ω is the matrix multiplication factor (≈ 2.373), using the idea of network coding. We provide in this paper an efficient algorithm that computes the edge connectivities $\lambda(u, v)$ for all pairs of nodes (u, v) such that $\lambda(u, v) \leq k$ in $O(kmn)$ time, for any integer $k \geq 1$. Our algorithm also computes a minimum u-v cut for each such pair of nodes (u, v).

Gomory-Hu Tree. It was observed by Gomory and Hu [7] long ago that the edge connectivities between all pairs of nodes in an undirected graph $G(V, E)$ can be represented by a weighted tree T on all nodes V such that

- the edge connectivity between any two nodes $u, v \in V$ equals the weight of the lightest edge on the unique u-v path in T.
- the partition of the nodes produced by removing this edge from T forms a minimum u-v cut in graph G.

Any tree satisfying both conditions is called a *cut-equivalent tree*, or *Gomory-Hu tree* of G; if a tree satisfies only the first condition, then it is called a *flow-equivalent tree* of G. The computation of a Gomory-Hu tree of any undirected graph can be reduced to the computation of n max-flows [7, 8], which yields a total running time of $\tilde{O}(mn^{1.5})$ using the current fastest unit capacity max-flow algorithm [11]. Currently the above running time is the best for any deterministic cut-equivalent tree construction. For randomized Gomory-Hu tree construction, Hariharan et al. [10] proposed an algorithm that with high probability computes a Gomory-Hu tree for any unweighted undirected graph in $\tilde{O}(mn)$ time.

The definition of Gomory-Hu tree and flow-equivalent tree can be naturally generalized to digraphs. Schnorr [13] attempted to construct the Gomory-Hu tree for general weighted digraphs. However, it was later pointed out by Benczur [1] that for general weighted digraphs, Gomory-Hu trees do not exist. We generalize the counter-example of Benczur [1] and show that the Gomory-Hu tree does not exist even in some unweighted digraph.

Fact 1 (Non-existence of Gomory-Hu tree for digraphs) *There exists an unweighted digraph that does not have any Gomory-Hu tree.*

Contrary to the Gomory-Hu tree, a flow-equivalent tree always exists for any weighted digraph. Cheng and Hu [2] generalized the result of Gomory and Hu [7] to construct a flow-equivalent tree using $O(n)$ computations of max-flows, which yields a total running time of $\tilde{O}(mn^{1.5})$. Actually Cheng and Hu proved something even more powerful: given any set V of n nodes, if we attach arbitrary weight $w(S)$ to each subset $S \subseteq V$ of nodes, the minimum weight cuts that separate all $\binom{n}{2}$ pairs of nodes can be represented by an *ancestor tree*, which is a tree spanning all nodes V.

We show in this paper that in directed unweighted graphs, the problem of computing the all-pairs edge connectivities and the computation of flow-equivalent tree are highly related to each other. By the following theorem, the result of Cheng and Hu [2] and that of Cheung et al. [3] hold for both problems.

Theorem 1 (Reducibility). *For any digraph G with n nodes, the all-pairs edge connectivities problem and the flow-equivalent tree problem are $O(n^2)$-reducible*

- *given the edge connectivities $\lambda(u,v)$ of all pairs of (u,v), a flow-equivalent tree of G can be constructed in $O(n^2)$ time.*
- *given a flow-equivalent tree of G, the edge connectivities $\lambda(u,v)$ of all pairs of (u,v) can be computed in $O(n^2)$ time.*

Low Edge Connectivities. In many applications, computing the edge connectivities of pairs of nodes which are poorly connected in the graph is more important. In particular, we consider the problem of computing the edge connectivities for the pairs of nodes (u,v) whose edge connectivities are at most k in the input graph, for any integer $k \geq 1$. Using the same definition from Hariharan et al. [9], the output should be represented succinctly as a weighted tree T whose nodes are $V_1, V_2, \ldots, V_l$, a partition of V, with the property that (1) for all $i \in [l]$, $\lambda(u,v) > k$ for all $u,v \in V_i$; (2) edge connectivity between $u \in V_i$ and $v \in V_j$, if $i \neq j$, is equal to the weight of the lightest edge in the unique V_i-V_j path in T. We call the above weighted tree a *k-edge-connectivity tree*. Note that for $k \geq \Delta = \max_{u,v \in V} \lambda(u,v)$, the k-edge-connectivity tree is a flow-equivalent tree. The problem for undirected graphs was considered by Hariharan et al. [9], who presented a randomized algorithm with expected running time $\tilde{O}(m+nk^3)$.

We consider in this paper the same problem in digraphs. For the special case when $k = 1$, Georgiadis et al. [6] showed that the 1-edge-connectivity tree can be constructed in linear time. However, for general k, the best algorithm to solve this problem involves computing all-pairs edge connectivities, which requires $\tilde{O}(mn^{1.5})$ time by Cheng and Hu [2]. We improve in this paper the above result to $O(kmn)$. It is easy to verify from our proofs that the following result holds even in directed multigraphs (in which case $m = \omega(n^2)$ is possible).

Theorem 2 (Computing low edge connectivities). *Given a digraph $G(V,E)$ and an integer $k \geq 1$, a k-edge-connectivity tree of G can be computed in $O(kmn)$ time.*

While it is shown by Cheng and Hu that the $\binom{n}{2}$ edge connectivities can be computed using $O(n)$ computations of max-flows, improving their running time in the low edge connectivity case is non-trivial. As we compute $\lambda(u,v)$, we actually obtain a minimum u-v cut, which defines a partition of nodes and this piece of information can be reused in the computation of the edge connectivities of other pairs of nodes. The above observation is crucial for Cheng and Hu's algorithm. However, in the low edge connectivity problem, if $\lambda(u,v) \geq k$, then we can not afford to compute a minimum u-v cut.

Instead, we decompose the computation of edge connectivities such that for each pair of nodes (u,v), the lower bound for $\lambda(u,v)$ is increased by 1 (if possible)

in each iteration. We maintain partitions of nodes in our algorithm and attach a *seed* node to each partition. Using the seeds to represent the edge connectivities between nodes in the same partition and a crucial *merge-flow* subroutine, we are able to reduce the total computation time in each iteration to $O(mn)$, which directly yields Theorem 2.

2 Preliminaries

Given a subset $S \subseteq V$ of nodes, let $d_-(S) = |\{(u,v) \in E | u \in S, v \in V \backslash S\}|$ be the *out-degree* of S, $d_+(S) = |\{(u,v) \in E | u \in V \backslash S, v \in S\}|$ be the *in-degree* of S and $d(S) = \min\{d_-(S), d_+(S)\}$ be the *degree* of S.

Definition 1 (Edge-Connectivity). *Given $u, v \in V$, the* edge-connectivity $\lambda(u,v)$ *between u and v is the minimum number of edges whose removal makes u and v not strongly connected. We assume $\lambda(u,u) = \infty$ for all $u \in V$.*

Given two nodes $u, v \in V$, we use $f(u,v)$ to denote the max-flow from u to v (assume that unit capacity is attached to each directed edge in the graph). By the above definition, we have $\lambda(u,v) = \min\{f(u,v), f(v,u)\} = \lambda(v,u)$. Moreover, there must exist at least one $S \subsetneq V$ such that $u \in S$, $v \in V \backslash S$ and $d(S) = d(V \backslash S) = \lambda(u,v)$. By Menger's Theorem, we have the following basic fact.

Fact 2 (i-Edge-Connectivity is an Equivalence Relation) *For any integer $i \geq 1$, given any nodes $a, b, c \in V$ such that $\lambda(a,b) \geq i$ and $\lambda(b,c) \geq i$, we have $\lambda(a,c) \geq i$.*

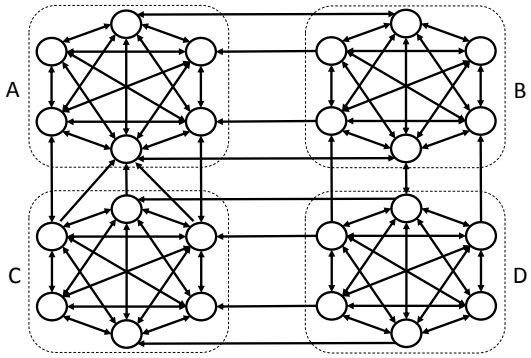
Throughout this paper, we use n to denote the number of nodes, m for the number of edges and $\Delta = \max_{u,v \in V, u \neq v} \lambda(u,v)$ to denote the maximum edge-connectivity between any two nodes in V. Unless otherwise stated, a graph is always directed and unweighted.

Definition 2 (Blocks, Partition). *Given a graph $G(V,E)$, for any integer $i \geq 0$, an* i-edge-connected (i-ec) block *is a subset of nodes $B \subseteq V$ such that $\forall u, v \in B$, $\lambda(u,v) \geq i$ and $\forall u \in B, v \in V \backslash B$, $\lambda(u,v) < i$. An* i-edge-connected partition*, denoted by Ω_i, is the collection of all i-edge-connected blocks of G.*

Definition 3 (Seed). *We attach a unique* seed $r(B) \in B$ *to each block B. For any $(i+1)$-ec block B' that is a subset of a i-ec block B, if $r(B) \in B'$, then we set $r(B') = r(B)$.*

By definition, we have $\Omega_0 = \{V\}$ and $\Omega_{\Delta+1} = \{\{u\} | u \in V\}$. Note that $\cup_{i=0}^{\Delta+1} \Omega_i$ is a Laminar family and hence we can organize all the blocks by a tree rooted at V such that block B is the *parent block* of block $B' \in \Omega_i$ iff $B' \subseteq B$ and $B \in \Omega_{i-1}$. We call B' a *child block* of B. If $r(B') = r(B)$, then we call B' the *closest* child block of B. Note that if $u \in V$ is the seed of some block in Ω_i, then u will be a seed for exactly one block in Ω_j, for each $j = i+1, i+2, \ldots, \Delta+1$.

Proof of **Fact 1**: The following graph does not have a Gomory-Hu tree, where "$\leftrightarrow$" stands for two directed edges in two directions. It is easy to see that the given graph has four 5-edge-connected blocks: A, B, C, D.



Suppose the given graph has a Gomory-Hu tree T, then if we contract all edges of weight at least 5 in T, then all nodes in the same 5-edge-connected block should become a single super-node. Hence in the contracted tree, there are only four super-nodes: u_A, u_B, u_C, u_D, each of them corresponds to a block.

It can be easily verified that $\forall a \in A$, we have: $\forall d \in D, \lambda(a,d) = 1$, $\forall b \in B, \lambda(a,b) = 2$, $\forall c \in C, \lambda(a,c) = 3$. Hence the four super-nodes are connected by three edges of weight at most 3. However, in the contracted tree, u_A, u_B and u_C can not be leaf nodes: if u_A is a leaf node, then after removing the edge connected to u_A in the contracted tree from T, the nodes will be partitioned into two sets A and $B \cup C \cup D$ and we have $d(A) = 4$, which is a contradiction since the edge connecting u_A should be of weight at most 3; similarly, u_B and u_C can not be leaf nodes since $d(C) \geq d(B) \geq 5$. Since any tree must have at least two leaf nodes, we conclude that G does not have any Gomory-Hu tree. □

Throughout this paper, we will use Tarjan's $O(m)$ time algorithm [14] for computing the strongly connected components of a digraph as a subroutine.

3 Flow-Equivalent Tree for Directed Graph

We first show that a k-edge-connectivity tree can be efficiently constructed given the partitions Ω_1, $\Omega_2, \ldots, \Omega_k, \Omega_{k+1}$. The following lemma is important for the proofs of Theorem 1 and Theorem 2.

Lemma 1 (Partitions to k-edge-connectivity tree). *For any integer $k \geq 1$, given partitions Ω_1, $\Omega_2, \ldots, \Omega_{k+1}$ of graph $G(V,E)$, a k-edge-connectivity tree of G can be constructed in $O(kn)$ time.*

Proof. Let S be the set of seeds for blocks in Ω_{k+1}. Note that we only need to build a tree spanning all nodes in S since (1) $\forall u, v \in S$, we have $\lambda(u,v) \leq k$; (2) $\forall u \in V$, there exists $r \in S$ s.t. $\lambda(u,r) > k$. We initialize the flow-equivalent tree $T = (S, \emptyset)$ and add edges to the tree as follows.

For $i = k, k-1, \ldots, 1, 0$, for each $B \in \Omega_i$ with at least 2 child blocks, connect the seeds of the child blocks (which must be in S) by a single path in T. Set the weight of each edge in the path to be i. By the above construction, it is easy to check that after considering each i we have

- all edges added to the current flow-equivalent tree are of weight at least i.
- two seeds r_1 and r_2 are connected iff $\lambda(r_1, r_2) \geq i$ (can be easily proved by induction on i).

Hence, every time when we add an edge (u, v) to the flow-equivalent tree, u and v must not be connected and the weight of the edge is set to be $\lambda(u, v)$, which means that after the whole construction, T is tree spanning S and for any two seeds u and v, the minimum weight of edges in the unique path between u and v equals $\lambda(u, v)$.

Replacing each $r \in S$ in the tree by the $(k+1)$-ec block B such that $r(B) = r$ gives a k-edge-connectivity tree. Note that when considering Ω_i, we only need to scan each node once. Hence the total running time for constructing the flow-equivalent tree is $O(kn)$. □

Using Lemma 1 with $k = \Delta$, we are able to prove Theorem 1. Recall that a Δ-edge-connectivity tree is a flow-equivalent tree.

Proof of **Theorem 1**: First, given a flow-equivalent tree $T(V, E_T)$ of $G(V, E)$, we can recover the edge-connectivities $\lambda(u, v)$ for all pairs of nodes as follows. Sort the edges in E_T by their weights in non-decreasing order in $O(n \log n)$ time. In every step we remove the edge e with minimum weight w_e from T, and set the edge connectivity $\lambda(u, v) = w_e$ for each pair of (u, v) that are disconnected in T by the removal of e. Hence the total running time can be bounded by $T(n) = \max_{i \in [n]}\{i(n-i) + T(i) + T(n-i) + O(n)\}$, which can be bounded by $O(n^2)$, using mathematical induction on $n \geq 1$.

By Lemma 1, to show that a flow-equivalent tree can be constructed given the edge-connectivities between all pairs of nodes, we only need to construct $\Omega_1, \Omega_2, \ldots, \Omega_\Delta$ of graph $G(V, E)$ in $O(n^2)$ time.

Algorithm 1 all-partitions($G(V, E)$):

```
1: let Ω_0 = {V}, set r(V) to be an arbitrary node in V.
2: for each t = 1, 2, ..., Δ do
3:     Ω_t = ∅, i = 1.
4:     for each B ∈ Ω_{t-1} do
5:         B_i = {v ∈ B | λ(r(B), v) ≥ t}, r(B_i) = r(B).
6:         Ω_t = Ω_t ∪ {B_i}, i = i + 1.                    ▷ O(|B|) time
7:         while ∪_{j=1}^{i-1} B_j ≠ B do
8:             pick an arbitrary u ∈ B \ ∪_{j=1}^{i-1} B_j.
9:             B_i = {v ∈ B | λ(u, v) ≥ t}, r(B_i) = u.
10:            Ω_t = Ω_t ∪ {B_i}, i = i + 1.                ▷ O(|B|) time
```

Given the edge-connectivities of all pairs of nodes, the above algorithm constructs each Ω_t in $O(\max\{1, |\Omega_t| - |\Omega_{t-1}|\}n)$ time for all $t = 1, 2, \ldots, \Delta$. Hence all partitions $\Omega_1, \Omega_2, \ldots, \Omega_\Delta$ of graph $G(V, E)$ can be constructed in $O(n^2)$ time, which by Lemma 1 means that a flow-equivalent tree can be constructed given the edge-connectivities between all pairs of nodes in $O(n^2)$ time. □

4 Computing Low Edge Connectivities

For any integer $k \geq 1$, we describe in this section how to compute the partitions $\Omega_1, \Omega_2, \ldots, \Omega_k, \Omega_{k+1}$ of G in $O(kmn)$ time.

Note that given the i-ec partition Ω_i, the $(i+1)$-ec partition Ω_{i+1} can be obtained by computing all child blocks of each i-ec block $B \in \Omega_i$. During each refinement step, we also need to assign seeds to the child blocks. The following algorithm applies the above steps to construct partitions $\Omega_1, \Omega_2, \ldots, \Omega_k, \Omega_{k+1}$, where function blocks$(B, r(B), r(B), i)$ returns all i-ec blocks of the $(i-1)$-ec block B.

Algorithm 2 partitions$(G(V,E),\ k)$:

fix any node $s \in V$, let $\Omega_0 = \{V\}$, $r(V) = s$.
for each $i = 1, 2, \ldots, k, k+1$ **do**
    $\Omega_i = \emptyset$.
    for each $B \in \Omega_{i-1}$ **do**
        $\Omega_i = \Omega_i \cup$ blocks$(B, r(B), r(B), i)$. ▷ partition B into i-ec blocks
return $\Omega_1, \Omega_2, \ldots, \Omega_k, \Omega_{k+1}$.

4.1 Block Refinement

Suppose that before constructing Ω_i, for each pair of nodes (u, v): (1) we have already computed a current flow $\mathcal{F}(u, v)$, which is stored as a set of edge-disjoint paths from u to v; (2) $|\mathcal{F}(u, v)| = \min\{\lambda(u, v), i-1\}$. Then for each $(i-1)$-ec pair of nodes (u, v), we try to find an augmenting path from u to v (from v to u) in the residual graphs with flow $\mathcal{F}(u, v)$ (flow $\mathcal{F}(v, u)$). If we can increase the flow by 1 in both directions, then u and v are at least i-ec and we place them into the same i-ec block. Otherwise we find a min-cut $(W, V \backslash W)$ with $d(W) = i-1$ that separates u and v in G and hence recursion can be applied.

Algorithm 3 blocks(B, r, u, i):

$S = \{u\}, r(S) = u, R = B \backslash S, \mathcal{B} = \emptyset$. ▷ if $u \neq r$, then $\forall v \in B$, $\lambda(v, r) = i-1$
while $R \neq \emptyset$ **do**
    pick an arbitrary node $v \in R$.
    if flow$(u, v, r, i) \neq$ NULL and flow$(v, u, r, i) \neq$ NULL **then** ▷ $O(m)$ time
        $S = S \cup \{v\}, R = R \backslash \{v\}$.
    else ▷ $\lambda(u, v) = i-1$
        if flow$(u, v, r, i) ==$ NULL **then**
            let $\tilde{G}$ be the residual graph with flow $\mathcal{F}(u, v)$.
        else
            let $\tilde{G}$ be the residual graph with flow $\mathcal{F}(v, u)$. ▷ $O(m)$ time
        let W be the set of nodes strongly connected to u in $\tilde{G}$.
        $\mathcal{B} = \mathcal{B} \cup$ blocks$(R \backslash W, r, v, i)$, $R = R \cap W$.
return $\mathcal{B} \cup \{S\}$.

The above algorithm takes a subset $B \subseteq B'$ of an $(i-1)$-ec block B', together with the seed $r = r(B')$ and a starting node $u \in B$, computes all i-ec blocks

that are subsets of B recursively. The function flow(u, v, r, i) computes a flow $|\mathcal{F}(u, v)| = i$ or returns NULL if $f(u, v) < i$.

Note that whenever we find a min-cut $(W, V \backslash W)$ with $d(W) = i - 1$ that separates u and v in G, then $\forall x \in B \backslash W$, we have $\lambda(u, x) = i - 1$, which means that $(W, V \backslash W)$ is a minimum u-x cut for all $u \in W$ and $x \in B \backslash W$. Hence we can recursively compute the i-ec blocks for $R \backslash W$ without splitting any i-ec block. Assume that flow(u, v, r, i) computes a flow $|\mathcal{F}(u, v)| = i$ or returns NULL if $f(u, v) < i$ (such an algorithm will be provided in Section 4.2), we have the following lemma immediately.

Lemma 2 (Block Refinement). *Given a subset $B \subseteq B'$ of an $(i-1)$-edge-connected block B' and the seed $r = r(B)$, Algorithm 3 returns all i-ec blocks that are subsets of B.*

While we assume that before constructing Ω_i, we have already computed a current flow $|\mathcal{F}(u, v)| = \min\{\lambda(u, v), i - 1\}$ between any pair of nodes, it is easy to observe that the time and space complexity is too large: in the worst case we need to update $\Theta(n^2)$ current flows when constructing one partition and store $O(mn^2)$ edges, which may be $\omega(m^2)$ already. Hence, we need to represent all $\Theta(n^2)$ current flows using a sparse structure, i.e., $O(n)$ current flows, such that the current flow between any pair of nodes can be efficiently recovered.

4.2 Computing the Current Flow with Seed Replacement

Note that in order to test i-edge-connectivity in an $(i-1)$-ec block B, we only need to do the test between every node $u \in B$ with the seed of B. Moreover, given any two nodes $u, v \in B$, if we have already computed $\mathcal{F}(u, r)$ and $\mathcal{F}(r, v)$ such that $|\mathcal{F}(u, r)| = i$ and $|\mathcal{F}(r, v)| = i$, then we can recover a flow $\mathcal{F}(u, v)$ with $|\mathcal{F}(u, v)| = i$ from u to v in $O(m)$ time using the following algorithm. We regard a path P as a sequence of edges and use $|P|$ to denote the number of edges in P.

Lemma 3 (Merge-Flows). *Given $u, v \in B \in \Omega_i$, $r = r(B)$, $|\mathcal{F}(u, r)| = i$ and $|\mathcal{F}(r, v)| = i$, Algorithm 4 computes a flow $\mathcal{F}(u, v)$ with $|\mathcal{F}(u, v)| = i$ from u to v in $O(m)$ time.*

Proof. Let $P_1, P_2, \ldots, P_i$ be i edge-disjoint paths from u to r and $Q_1, Q_2, \ldots, Q_i$ be i edge-disjoint paths from r to v. Note that there might exist edges that are used by both the P_j's and the Q_l's (such a *shared* edge will be given two labels by Algorithm 4). We argue that we can compute a matching between the P_j's and the Q_l's in $O(m)$ time such that each P_j is matched with $Q_{\text{matchP}(j)}$ to form a new path H_j from u to v. Moreover, all H_j's are edge-disjoint. Then we have $\mathcal{F}(u, v) = \{H_1, H_2, \ldots, H_i\}$ as required.

In Algorithm 4, H_j is set to be $(p_{j,1}, p_{j,2}, \ldots, p_{j,\text{top}(j)}, q_{l,x+1}, q_{l,x+2}, \ldots, q_{l,|Q_l|})$, where $l = \text{matchP}(j)$, $p_{j,\text{top}(j)} = q_{l,x}$ is a shared edge (or $\text{top}(j) = |P_j|$ and $x = 0$). Note that by the end of the while loop, we have $\text{matchP}(j) \neq 0$ for all $j \leq i$ and we have formed a partial matching between P_j's and Q_l's: the number

Algorithm 4 merge-flow(u, v, r, i)

1: let $\mathcal{F}(u,r) = \{P_1, P_2, \ldots, P_i\}$, $\mathcal{F}(r,v) = \{Q_1, Q_2, \ldots, Q_i\}$ and $M = 0$.
2: **for** each $j = 1, 2, \ldots, i$ **do**
3: label edges in P_j as $(p_{j,1}, p_{j,2}, \ldots, p_{j,|P_j|})$.
4: label edges in Q_j as $(q_{j,1}, q_{j,2}, \ldots, q_{j,|Q_j|})$.
5: $\forall j \le i$, let $\text{top}(j) = 0$, $\text{head}(j) = 0$, $\text{matchP}(j) = 0$, $\text{matchQ}(j) = 0$.
6: **while** $M < i$ **do**
7: pick an arbitrary $j \le i$ s.t. $\text{matchP}(j) = 0$.
8: **if** $\text{top}(j) == |P_j|$ **then**
9: $\text{matchP}(j) = -1$, $M = M + 1$. ▷ finished if r is reached
10: **else**
11: $\text{top}(j) = \text{top}(j) + 1$.
12: **if** $p_{j,\text{top}(j)}$ has another label $q_{l,x}$ **then**
13: **if** $\text{matchQ}(l) == 0$ **then** ▷ match Q_l with P_j, at position x
14: $\text{matchQ}(l) = j$, $\text{head}(l) = x$.
15: $\text{matchP}(j) = l$, $M = M + 1$.
16: **else if** $x > \text{head}(l)$ **then**
17: $\text{matchP}(\text{matchQ}(l)) = 0$, $\text{matchP}(j) = l$.
18: $\text{matchQ}(l) = j$, $\text{head}(l) = x$.
19: set $\mathcal{F}(u,v) = \emptyset$. ▷ add i edge-disjoint paths in $\mathcal{F}(u,v)$
20: **for** each $j = 1, 2, \ldots, i$ **do**
21: **if** $\text{matchP}(j) > 0$ **then**
22: $l = \text{matchP}(j)$, $x = \text{head}(l)$.
23: $\mathcal{F}(u,v) = \mathcal{F}(u,v) \cup \{(p_{j,1}, p_{j,2}, \ldots, p_{j,\text{top}(j)}, q_{l,x+1}, q_{l,x+2}, \ldots, q_{l,|Q_l|})\}$.
24: **else** ▷ $\text{matchP}(j) = -1$
25: pick any $l \le i$ such that $\text{matchQ}(l) == 0$, set $\text{matchQ}(l) = j$.
26: $\mathcal{F}(u,v) = \mathcal{F}(u,v) \cup \{(p_{j,1}, p_{j,2}, \ldots, p_{j,|P_j|}, q_{l,1}, q_{l,2}, \ldots, q_{l,|Q_l|})\}$.
27: **return** $\mathcal{F}(u,v)$.

of P_j's with $\text{matchP}(j) = -1$ equals the number of Q_l's with $\text{matchQ}(l) = 0$. We can form an arbitrary matching between P_j's with $\text{matchP}(j) = -1$ and Q_l's with $\text{matchQ}(l) = 0$.

At any moment during the execution of Algorithm 4, we use $E_p = \cup_{j\in[i]}\{p_{j,1}, p_{j,2}, \ldots, p_{j,\text{top}(j)}\}$ to denote the set of "p-edges" that are already scanned and $E_q = \cup_{l\in[i]}\{q_{l,\text{head}(l)+1}, q_{l,\text{head}(l)+2}, \ldots, q_{l,|Q_l|}\}$. We show that $E_p \cap E_q = \emptyset$ during the whole execution.

Showing that $E_p \cap E_q = \emptyset$ is trivial before the while loop since $E_p = \emptyset$. During each while loop (line 7-18) we increase $\text{top}(j)$ by one, for some $j \le i$, which include one more edge e in E_p. If e is not shared, then it is safe to include e in E_p. Otherwise (line 13-18), assume $e = p_{j,\text{top}(j)} = q_{l,x}$. The algorithm makes sure that $\text{head}(l) \ge x$ at the end of this iteration of while loop, which exclude $e = q_{l,x}$ from E_q and maintains $E_p \cap E_q = \emptyset$. Hence we conclude that $E_p \cap E_q = \emptyset$ at the end of the whole while loop. Since the H_j's only use edges in E_p once and use edges in E_q once, all paths H_j's are edge-disjoint.

It is easy to check that each while loop can be executed in $O(1)$ time and increases the size of E_p by exactly one, the first part (line 1-18) of Algorithm 4 executes in $O(m)$ time. Since the execution time of the second part (line 19-27) of Algorithm 4 can be bounded by $O(|E_p \cup E_q|) = O(m)$, we conclude that the i edge-disjoint paths $\mathcal{F}(u, v)$ can be computed in $O(m)$ time by Algorithm 4. □

Hence instead of computing and storing $\Theta(n^2)$ current flows, when constructing Ω_i, we only need to know the current flow between nodes in the same $(i-1)$-ec block. Moreover, to represent the current flow between two nodes in the same block, we only need to store the current flows between the seed of the block and all other nodes in the block, which reduces the total number of current flows we need to update from $\Theta(n^2)$ to $O(n)$.

Algorithm 5 flow(u, v, r, i)

```
1: if u ≠ r and v ≠ r then                          ▷ min{λ(u,r), λ(v,r)} = i − 1
2:     merge-flow(u, v, r, i − 1).                  ▷ O(m) time
3: if v is reachable from u in the residual graph with flow F(u,v) then
4:     find an augmenting path from u to v in the residual graph.   ▷ O(m) time
5:     merge the path with i − 1 paths in F(u, v).
6:     return F(u, v).
7: else
8:     return NULL.
```

Lemma 4. *For all $i \geq 1$, after constructing Ω_{i-1}, we have*

1. *$\mathcal{F}(u, v)$ contains $i-1$ edge-disjoint paths from u to v if $u \in B \in \Omega_{i-1}, v = r(B)$ or $v \in B \in \Omega_{i-1}, u = r(B)$.*
2. *given $u, v \in B \in \Omega_{i-1}$ and $r = r(B)$, Algorithm 5 computes a flow $\mathcal{F}(u, v)$ with $|\mathcal{F}(u, v)| = i$ or returns NULL if $\lambda(u, v) = i - 1$ in $O(m)$ time.*

Proof. We prove the above statements by induction on $i \geq 1$.

The base case ($i = 1$) is trivial since (1) $\Omega_0 = \{V\}$ and $|\mathcal{F}(u, v)| = 0$ for all $u, v \in V$; (2) computing a path from u to v in G (if possible) can be done in $O(m)$ time. Now assume the statement is true for i and consider $i + 1$. By induction hypothesis, $\forall u \in B \in \Omega_i$, we have $|\mathcal{F}(u, r(B))| = |\mathcal{F}(r(B), u)| = i$. Moreover, for any two nodes u and v, line 4 of Algorithm 3 computes in $O(m)$ time i edge-disjoint paths from u to v and i edge-disjoint paths from v to u.

By Algorithm 3, it is easy to observe that $\cup_{S \in \mathcal{B}} S = B$, where $\mathcal{B}$ is the set of i-ec blocks returned after executing blocks(B, r, u, i). Note that in every execution of Algorithm 3, we increase the size of $\mathcal{B}$ by exactly one. Let S be the last i-ec block that is included into $\mathcal{B}$ in some execution of Algorithm 3. Hence to prove statement-(1) for $i+1$, we only need to show that for all $v \in S$, we have $|\mathcal{F}(r(S), v)| = |\mathcal{F}(v, r(S))| = i$, which is obvious since every node v is included in S only if v passes of the test in line 4.

Assuming that statement-(1) is true for $i+1$, proving statement-(2) is straightforward. By Lemma 3, line 2 of Algorithm 5 executes in $O(m)$ time and hence we can get a current flow $|\mathcal{F}(u, v)| = i - 1$ from u to v in $O(m)$ time. Then in $O(m)$ time, we can either increase the flow from u to v by one, or conclude that $\lambda(u, v) = i - 1$. □

4.3 Complexity of the Construction

We have described how to compute all child blocks of any block B using recursive algorithm (Algorithm 3) which uses a subroutine Algorithm 5 to compute a flow from u to v in $O(m)$ time. We will analyze the total running time for the construction of partitions $\Omega_1, \Omega_2, \ldots, \Omega_k, \Omega_{k+1}$ in this section.

Notice that given an $(i-1)$-ec block B, Algorithm 3 (run as blocks$(B, r(B), r(B), i)$) computes all child blocks of B by using Algorithm 5 $O(|B|)$ times. Moreover, we have the following important observations:

- in every call of Algorithm 5 (run as flow(u, v, r, i)), exactly one of u, v has been assigned to be the seed for an i-ec block.
- before every recursive call of Algorithm 3 (line 11), we compute the strongly connected components of the residual graph with flow $\mathcal{F}(u, v)$ (or $\mathcal{F}(v, u)$), where $\lambda(u, v) = i - 1$ and exactly one of u, v has been assigned to be the seed for an i-ec block.
- in every recursive call of Algorithm 3 (line 12), node v will be assigned to be the seed of some child block of B.

Charging argument. By the above observation, we can charge the running time $O(m)$ for computing a current flow from u to v using Algorithm 5 and the running time $O(m)$ for computing the strongly connected components to the non-seed node. Hence any node u will not be charged after being assigned to be the seed of some block by the above argument.

Lemma 5 (Running time on each node). *By the above charging argument, every node u will only be charged a total running time of $O(km)$.*

Proof. First observe that Algorithm 5 (run as flow(u, v, r, i) and flow(v, u, r, i)) is always called twice at a time (for computing flows $|\mathcal{F}(u, v)| = |\mathcal{F}(v, u)| = i$ from u to v and from v to u) and the computation cost will be charged to the same (non-seed) node. Suppose the node being charged is u, we say that u is charged *with requirement i* in the above case.

Note that for every node u, after being charged with requirement i, either u is placed in some i-ec block, or we conclude that $\lambda(u, v) = i - 1$. Note that a non-seed node in an i-ec block will not be charged with requirement i again. Moreover, if the requirement is not satisfied (flow(u, v, r, i) = NULL or flow(v, u, r, i) = NULL), then after being charged an extra $O(m)$ running time for computing the strongly connected components, u will be immediately assigned to be a seed.

Since all computation cost between u and v afterwards will be charged to the non-seed node v, we conclude that every node u will only be charged a total running time of $O(km)$. □

Proof of **Theorem 2**: To prove Theorem 2, it suffices to show that partitions $\Omega_1, \Omega_2, \ldots, \Omega_k, \Omega_{k+1}$ can be constructed in $O(kmn)$ time, by Lemma 1.

By Lemma 2, we can use Algorithm 3 (block$(B, r(B), r(B), i)$) to compute all the child blocks of any block B. Hence Algorithm 2 correctly computes partitions $\Omega_1, \Omega_2, \ldots, \Omega_k, \Omega_{k+1}$. Thus we only need to bound the running time.

By Lemma 5, the total running time charged on nodes is at most $O(kmn)$. Since the total running time not charged on any node, i.e., while loops and for loops, for the computation of partitions can be bounded by $O(\Delta n)$, we conclude that the computation of partitions $\Omega_1, \Omega_2, \ldots, \Omega_k, \Omega_{k+1}$ (and hence the k-edge-connectivity tree) of any graph G can be done in $O(kmn)$ time. □

References

1. András A Benczúr. Counterexamples for directed and node capacitated cut-trees. *SIAM Journal on Computing*, 24(3):505–510, 1995.
2. Chung-Kuan Cheng and T. C. Hu. Ancestor tree for arbitrary multi-terminal cut functions. In *Proceedings of the 1st Integer Programming and Combinatorial Optimization Conference, Waterloo, Ontorio, Canada, May 28-30 1990*, pages 115–127, 1990.
3. Ho Yee Cheung, Lap Chi Lau, and Kai Man Leung. Graph connectivities, network coding, and expander graphs. In Rafail Ostrovsky, editor, *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 190–199. IEEE Computer Society, 2011.
4. EA Dinits. Algorithm of solution to problem of maximum flow in network with power estimates. *Doklady Akademii Nauk SSSR*, 194(4):754, 1970.
5. Shimon Even and R Endre Tarjan. Network flow and testing graph connectivity. *SIAM journal on computing*, 4(4):507–518, 1975.
6. Loukas Georgiadis, Giuseppe F. Italiano, Luigi Laura, and Nikos Parotsidis. 2-edge connectivity in directed graphs. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1988–2005. SIAM, 2015.
7. Ralph E Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial & Applied Mathematics*, 9(4):551–570, 1961.
8. Dan Gusfield. Very simple methods for all pairs network flow analysis. *SIAM Journal on Computing*, 19(1):143–155, 1990.
9. Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. Efficient algorithms for computing all low st edge connectivities and related problems. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 127–136. Society for Industrial and Applied Mathematics, 2007.
10. Ramesh Hariharan, Telikepalli Kavitha, Debmalya Panigrahi, and Anand Bhalgat. An o (mn) gomory-hu tree construction algorithm for unweighted graphs. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 605–614. ACM, 2007.
11. Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in õ (vrank) iterations and faster algorithms for maximum flow. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 424–433. IEEE, 2014.
12. Karl Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.
13. Claus-Peter Schnorr. Bottlenecks and edge connectivity in unsymmetrical networks. *SIAM Journal on Computing*, 8(2):265–274, 1979.
14. Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.