Computing Vertex Connectivity: New Bounds from Old Techniques

Monika R. Henzinger * Satish Rao† Harold N. Gabow ‡

August 22, 1999

Abstract

The vertex connectivity $\boldsymbol{\kappa}$ of a graph is the smallest number of vertices whose deletion separates the graph or makes it trivial. We present the fastest known deterministic algorithm for finding the vertex connectivity and a corresponding separator. The time for a digraph having n vertices and m edges is $O(\min\{\boldsymbol{\kappa}^3 + n, \boldsymbol{\kappa} n\}m)$; for an undirected graph the term m can be replaced by $\boldsymbol{\kappa} n$. A randomized algorithm finds $\boldsymbol{\kappa}$ with error probability 1/2 in time $O(nm)$. If the vertices have nonnegative weights the weighted vertex connectivity is found in time $O(\boldsymbol{\kappa}_1 nm \log(n^2/m))$ where $\boldsymbol{\kappa}_1 \leq m/n$ is the unweighted vertex connectivity, or in expected time $O(nm \log(n^2/m))$ with error probability 1/2. The main algorithm combines two previous vertex connectivity algorithms and a generalization of the preflow-push algorithm of Hao and Orlin [13] that computes edge connectivity.

1 Introduction

The *(vertex) connectivity* $\boldsymbol{\kappa}$ of an undirected graph or digraph is the smallest number of vertices whose deletion separates or trivializes the graph. (Most terms are defined precisely at the end of this section.) This is a central concept of graph theory [15]. We present efficient algorithms for computing connectivity.

More precisely we consider the following tasks. To *compute the connectivity* means to find $\boldsymbol{\kappa}$ and a corresponding separator. To *check k-connectedness* means to verify that $\boldsymbol{\kappa} \geq k$ or find a separator of $< k$ vertices. Lastly if each vertex has a nonnegative weight, $\boldsymbol{\kappa}$ is the smallest weight of a separator or trivializor and we may wish to compute this weighted connectivity. We present algorithms for all these tasks, on undirected graphs and digraphs. Our approach is to simply combine three previous algorithms, two for computing vertex connectivity and one for network flow. We now discuss the efficiency and compare it to previous work. Throughout this paper n and m denote the number of vertices and edges of the given graph, respectively.

First consider (unweighted) digraphs. Even and Tarjan showed how to compute $\boldsymbol{\kappa}$ in time $O(\boldsymbol{\kappa} n^{1.5} m)$ [7]. This was improved to the following best-known bounds: Even [6] showed how to

*Compaq Systems Research, 130 Lytton Ave., Palo Alto, CA 94301. This research was supported in part by an NSF CAREER Award, Grant No. CCR-9501712. e-mail: monika@pa.dec.com

†Computer Science Division, Soda Hall, University of California, Berkeley, CA 94720-1776. e-mail: satishr@cs.berkeley.edu

‡Department of Computer Science, University of Colorado at Boulder, Boulder, CO 80309. e-mail: hal@cs.colorado.edu

check k-connectedness by solving at most $k^2 + n$ network flow problems, achieving a running time of $O((k + \sqrt{n})k\sqrt{n}m)$. Galil extended Even's approach to compute $\boldsymbol{\kappa}$ in the same time bound with k replaced by $\boldsymbol{\kappa}$ [8]. We find the connectivity in time $O(\min\{\boldsymbol{\kappa}^3 + n, \boldsymbol{\kappa}n\}m)$ and check k-connectedness in the same bound with $\boldsymbol{\kappa}$ replaced by k. This bound equals the previous best when k or $\boldsymbol{\kappa}$ is $\Theta(1)$ or $\Theta(\sqrt{n})$ and is superior for all other values. The largest improvement is a factor $\sqrt{n}$ when k or $\boldsymbol{\kappa}$ is $\Theta(n)$. Our algorithm uses $O(m)$ space, as do all other algorithms of this paper. Our high-level algorithm is essentially the same as [7] for small connectivities and [6, 8] for large connectivities.

Next consider undirected graphs. An undirected graph has the same connectivity as the digraph formed by giving each edge both directions. Hence the digraph algorithms apply. Furthermore the technique of Nagamochi and Ibaraki [17] usually allows one to work on a subgraph of $O(kn)$ edges rather than the whole graph. This usually allows undirected connectivity and k-connectedness to be computed in the digraph time bound with m replaced by kn or $\boldsymbol{\kappa}n$. This is true for all the above algorithms. For example we compute undirected connectivity in time $O(\min\{\boldsymbol{\kappa}^3+n, \boldsymbol{\kappa}n\}\boldsymbol{\kappa}n)$. We also mention a recent algorithm of Henzinger: In an undirected graph with minimum degree δ it finds $\boldsymbol{\kappa}$ and a corresponding separator if $\boldsymbol{\kappa} \leq \delta/2$ and verifies $\boldsymbol{\kappa} > \delta/2$ otherwise, in time $O(\min\{\boldsymbol{\kappa}, \sqrt{n}\}n^2)$ [12].

We turn to randomized algorithms. Becker et al. give a Monte Carlo version of [7] that computes the connectivity of a digraph. For any desired probability $p \leq 1/n$ it returns the correct answer with probability $1 - p$ (i.e., the error probability is p) and runs in expected time $O(((\log 1/p)/(\log (n/\boldsymbol{\kappa})))n^{1.5}m)$ [3]. We give an algorithm that computes the connectivity with error probability $1/2$ in (worst-case) time $O(nm)$. This is a strict improvement of [3] since for any desired probability p repeating our algorithm $\log 1/p$ times achieves error probability p in time $O((\log 1/p)nm)$. For undirected graphs the time to achieve error probability $1/2$ is $O(\boldsymbol{\kappa}n^2)$ (replace $\boldsymbol{\kappa}$ by k for checking k-connectedness). These results use a fact about our deterministic algorithm for computing $\boldsymbol{\kappa}$: It runs in time $O(((n - \boldsymbol{\kappa})n)^2)$. This bound comes into play when $\boldsymbol{\kappa} = n - o(n)$ (e.g., the time to compute $\boldsymbol{\kappa}$ is linear when $\boldsymbol{\kappa} = n - O(1)$).

Linial et al. check undirected k-connectedness with a Monte Carlo algorithm that runs in time $O((M(n) + nM(k)) \log n)$ and has error probability $1/n$ [16]. Here $M(n)$ is the time to multiply two $n \times n$ matrices and is $O(n^{2.38})$ [5]. Our Monte Carlo algorithm (for the same error probability) is faster when $k \leq n^{.37}$ or $k \geq n^{.73}$; it is always faster if naive matrix multiplication is used to get a practical algorithm. Linial et al. also give a Las Vegas algorithm with expected running time k times their Monte Carlo bound. Cheriyan and Reif achieve similar Monte Carlo and Las Vegas time bounds for digraphs [4]. Our deterministic algorithm is faster than these Las Vegas algorithms if they use naive matrix multiplication.

Finally consider vertex-weighted graphs (undirected or digraphs). The naive algorithm computes the weighted connectivity by calculating $\Theta(n^2)$ maximum flows. We know no other results on weighted connectivity. We improve this to time $O(\boldsymbol{\kappa}_1 nm \log (n^2/m))$ where $\boldsymbol{\kappa}_1$ denotes the connectivity if vertex weights are ignored (i.e., every vertex has weight 1). Since $\boldsymbol{\kappa}_1 \leq m/n$ this bound is at most $O(m^2 \log (n^2/m))$. This improves the naive bound by a factor greater than n if we use the best-known algorithm of [14] to compute a maximum flow. We also present a Monte Carlo algorithm that computes the connectivity with error probability 1/2 in expected time $O(nm \log(n^2/m))$.

The vertex connectivity algorithms we use transform the problem to a number of network flow problems, e.g., $\boldsymbol{\kappa} n$ maximum flow problems for [7]. We show the preflow-push approach to maximum flow can reduce the number of flow problems, e.g., to $\boldsymbol{\kappa}$ flow problems for [7]. The preflow-push approach is due to Goldberg and Tarjan [10]. Gallo, Grigoriadis and Tarjan show how this approach often allows n or more related maximum flow computations to be combined into one [11]. Hao and Orlin [13] use their idea to give an efficient algorithm to find the edge connectivity of a (capacitated) digraph in time $O(nm \log(n^2/m))$. We present a flow algorithm that extends [13]. In fact [13] sketches an extension but omits some details. To clarify this we first define our flow problem. The complete definition of the VC split problem is given at the end of this section.

Consider a digraph $G = (V, E)$ with a set of "terminals" $T \subseteq V$. A "minimum split for T" is a set $S \subseteq V$ that minimizes the total capacity of the entering edges subject to the constraint that both $T \cap S$ and $T - S$ are nonempty. The edge connectivity algorithm of [13] finds a minimum unrestricted cut, or in our terminology, a minimum split for V. We extend this algorithm to find a minimum split for T when T is a given vertex cover of G; we call this a "minimum VC split." We find a minimum VC split in the same time bound as the edge connectivity algorithm of [13] (e.g., for the above unweighted connectivity algorithms this bound is $O(nm)$).

In hindsight we discovered that Hao and Orlin sketch a similar extension of their algorithm, to find a minimum cut in a bipartite digraph ([13], Theorem 9, p.443; vertex connectivity is not discussed). They indicate that using the bipush variant of the preflow-push algorithm [2] correctly implements the extended algorithm. We do not use bipushes for our main VC split algorithm, but we do introduce a new operation called "join." Our join operation can decrease distance labels by large amounts; previous preflow-push algorithms never decrease distance labels and this is crucial for their efficiency [1, 2, 10, 11, 13, 14]. We prove that the desired time bounds hold in spite of this decrease. It is possible to implement joins slightly less efficiently so they do not decrease distances, but then they increase distances by large amounts. It seems that the phenomenon handled by joins must be addressed and analyzed in any complete algorithm for this problem. Thus our contribution is to fill a gap in the fundamentally sound and insightful suggestion of Hao and Orlin.

We anticipate further applications of the VC split algorithm besides vertex connectivity. For instance taking the vertex cover $T = V$ shows our algorithm generalizes [13], i.e., it can compute the edge connectivity in the same bound as [13]. The vertex cover $T = V - \{s\}$ implements Frank's digraph connectivity augmentation algorithm in the same time bound as [9].

The paper is organized as follows. Section 2 gives algorithms for vertex connectivity, assuming an efficient algorithm for finding a minimum VC split. Section 3 supplies the VC split algorithm. The rest of this section gives notation and definitions.

$\mathbf{R}_+$ denotes the set of nonnegative real numbers. We often denote singleton sets by omitting set braces, e.g., $U \cup u$. If $U, W \subseteq V$ then a $\overline{U}W$-*set* is a subset of V containing W and disjoint from U. For example for $u, w \in V$, a $\overline{u}w$-set contains w but not u. If f is a function $f : S \to \mathbf{R}$ then for any set $T \subseteq S$, $f(T)$ denotes $\sum\{f(t) : t \in T\}$.

In a graph with vertices v and w, the notation vw denotes an undirected edge joining v and w

or a directed edge from v to w; it will be clear from context which is meant. For a digraph G, G^R denotes the reverse digraph, i.e., all edges of G are reversed. The *undirected version* of G ignores the directions of edges.

A (directed) edge uv *enters* any $\overline{u}v$-set and *leaves* any $\overline{v}u$-set. For a set of vertices W, the *in-degree* $\rho(W)$ equals the total number of edges entering W. If the graph has a capacity function on the edges then $\rho(W)$ is the total capacity of edges entering W. An *xy-cut* is the set of edges entering some $\overline{x}y$-set. A *minimum xy-cut* is an xy-cut of smallest total capacity; this minimum equals the value of a maximum xy-flow.

Consider a digraph or undirected graph $G = (V, E)$. For $x, y \in V$, $S \subseteq V - \{x, y\}$ is an *x,y-separator* if $G - S$ contains no path from x to y. Define $\boldsymbol{\kappa}(x, y)$ as the smallest cardinality of an x, y-separator if such separators exist; if not (i.e., $x = y$ or $xy \in E$) then set $\boldsymbol{\kappa}(x, y) = n - 1$. Define the *vertex connectivity* by

$$\boldsymbol{\kappa} = \min\{\boldsymbol{\kappa}(x, y) : x, y \in V\}.$$

A *vertex-weighted graph* has a function $w : V \to \mathbf{R}_+$. In this context define $\boldsymbol{\kappa}(x, y) = \min\{w(S), \infty : S \text{ an } x, y\text{-separator}\}$. The *weighted vertex connectivity* $\boldsymbol{\kappa}$ is again defined by the above equation. For w identically 1 this is just ordinary vertex connectivity.

For an undirected graph $G = (V, E)$ the algorithm of [17] partitions E, in $O(m)$ time, into a sequence of forests F_k, $k = 1, \ldots, n$. For $k = 1, \ldots, n$ define the "forest subgraph" FG_k by

$$FG_k = (V, \cup_{i=1}^k F_i).$$

For all k, FG_k is k-connected if G is. In fact [17] proves the stronger property that any vertex separator of FG_k with cardinality $< k$ is a vertex separator of G (see the proof of Theorem 3.1 in [17]). Clearly FG_k has $O(kn)$ edges.

Consider a digraph $G = (V, E)$ with a set of "terminals" $T \subseteq V$. A subset of V is a *split of T* if it is an $\overline{s}t$-set for some vertices $s, t \in T$. If each edge has a nonnegative capacity, a *minimum split of T* is a split S of T that minimizes $\rho(S)$. For any $s \in T$, an *s-split of T* is an $\overline{s}t$-set for some $t \in T$, and we similarly define a *minimum s-split of T*. Clearly a minimum split of T is a minimum s-split of T in either G or G^R. The *minimum VC split problem* is to find a minimum s-split of T, a given vertex cover of (the undirected version of) G.

If a function refers to a graph we include the graph as an extra argument if it is not clear, e.g., $\rho(W, G)$, $\boldsymbol{\kappa}(G)$.

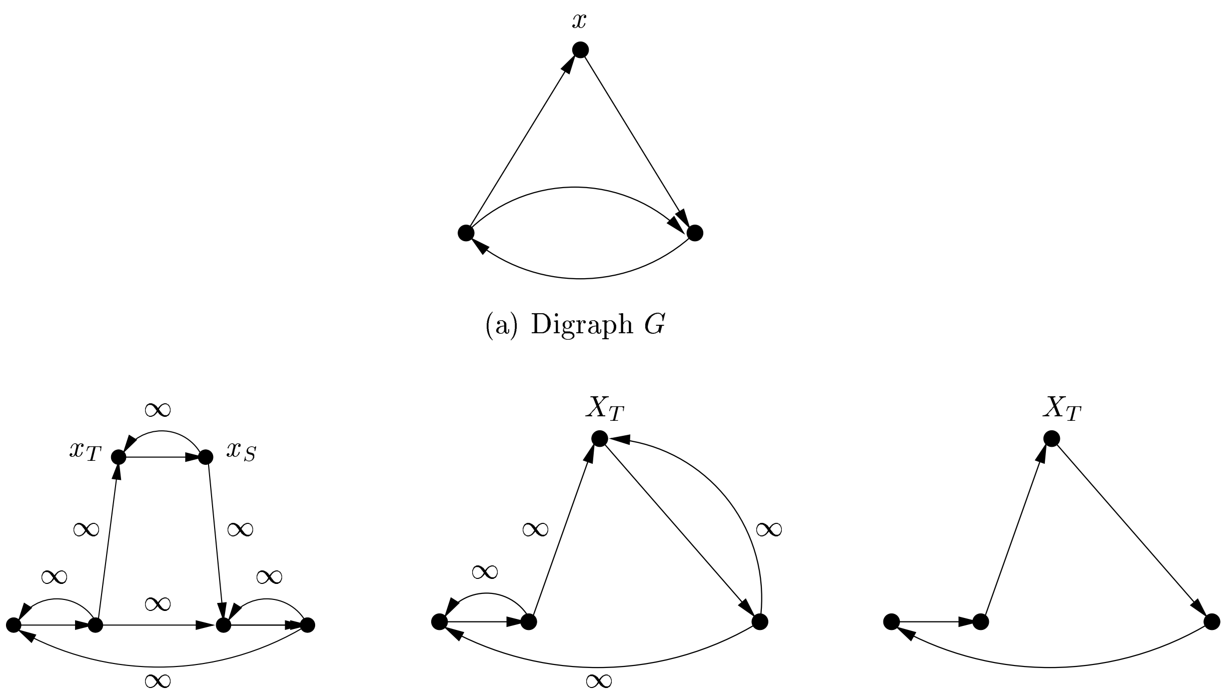
2 Vertex Connectivity

This section gives our algorithms for vertex connectivity. It summarizes their efficiency, assuming the results of Section 3 for the minimum VC split problem.

Consider a digraph $G = (V, E)$. For any $x \in V$ define

$$\boldsymbol{\kappa}(x) = \min\{\boldsymbol{\kappa}(x, y) : y \in V\}.$$

Clearly $\boldsymbol{\kappa} = \min\{\boldsymbol{\kappa}(x) : x \in V\}$. We begin by showing that computing $\boldsymbol{\kappa}(x)$ reduces to the minimum VC split problem. Assuming Theorem 3.2 this gives an algorithm to compute $\boldsymbol{\kappa}(x)$ in time $O(nm)$. In this discussion fix the digraph G and the vertex x. We use three graphs derived from G, illustrated in Fig. 1.



(a) Digraph G

(b) Split graph SG (c) Contracted graph CG (d) Unit capacity graph UG

Figure 1: A digraph (a) and its derived graphs (b)–(d). In (b)–(d) infinite capacity edges are labelled and all other edges have unit capacity.

We start with the standard reduction to network flow by node splitting. Specifically define a graph SG (the "split graph") having for each $v \in V$, two vertices v_T, v_S and edges $v_T v_S$ and $v_S v_T$ of capacity 1 and ∞ respectively, and for each edge $vw \in E$ an edge $v_S w_T$ of capacity ∞. There is a bijection between x, y-separators of G and finite $x_S y_T$-cuts of SG, specifically separator $U \subseteq V$ corresponds to cut $\{u_T u_S : u \in U\}$. Hence $\boldsymbol{\kappa}(x, y; G)$ equals the minimum capacity of an $x_S y_T$-cut in SG. Letting y be arbitrary we conclude that $\boldsymbol{\kappa}(x) = \boldsymbol{\kappa}(x, G)$ equals the capacity of a smallest $x_S y_T$-cut for a vertex y_T in SG, or $n-1$ if no such finite cut exists. Any such cut corresponds to an x, y-separator of $\boldsymbol{\kappa}(x)$ vertices (using the above bijection).

To find this cut using the algorithm of Section 3 it is convenient to use two other graphs. CG (the "contracted graph") is derived from SG by contracting x_S with its neighbors, i.e., contract vertices x_S, x_T and z_T for $z \in V$ with $xz \in E$. Call the vertex resulting from this contraction X_T. Assume X_T is not the only vertex of CG, since otherwise $\boldsymbol{\kappa}(x) = n - 1$. Observe the following:

> (i) $\boldsymbol{\kappa}(x)$ equals the capacity of the smallest $X_T y_T$-cut, for y_T a vertex of CG.
> (ii) Any such cut corresponds to an x, y-separator of $\boldsymbol{\kappa}(x)$ vertices.
> (iii) $C = \{y_T : y_T \text{ a vertex of } CG\}$ is a vertex cover of CG (this set includes X_T).

(i) holds since CG is constructed by contracting infinite capacity edges incident to the source x_S of SG. (iii) holds since C is one side of a bipartition of CG.

Next we transform CG to a unit capacity graph UG: Delete all edges $v_S v_T$ and assign capacity 1 to all edges $v_S w_T$. The resulting graph UG satisfies properties (i) and (iii). For (i) the transformation does not change the value of a maximum flow from X_T to any y_T, so it does not change the value of a minimum cut. ((ii) need not hold since new minimum cuts may be introduced.)

(i) and (iii) show that $\boldsymbol{\kappa}(x)$ is the value of a minimum X_T-split of vertex cover C in digraph UG (recall the definition of minimum s-split of T at the end of Section 1). Since UG is a graph with unit edge capacities, Theorem 3.2 shows $\boldsymbol{\kappa}(x)$ can be computed in time $O(nm)$. (Note that n and m for UG are within a constant factor of the same parameters for G.) It is a simple matter to derive an x, y-separator of $\boldsymbol{\kappa}(x)$ vertices from a minimum X_T-split: If the set of edge entering the minimum X_T-split is $\{u_T u_S : u \in U\} \cup \{v_S w_T : vw \in F\}$ (for some sets $U \subseteq V$, $F \subseteq E$) the desired separator can be taken as $U \cup \{w : vw \in F \text{ for some vertex } v\}$.

Now we show how our algorithm for $\boldsymbol{\kappa}(x)$ can be used to carry out a number of connectivity computations. The first group of results is summarized as follows. Define three functions

$$T_1(k,n,m) = knm, \quad T_2(k,n,m) = (k^3+n)m, \quad T = \min\{T_1, T_2\}.$$

Note $T = T_1$ for $k \geq \sqrt{n}$ and $T = T_2$ for $k \leq \sqrt{n}$.

Theorem 2.1 *For a digraph, the connectivity $\boldsymbol{\kappa}$ can be found in time $O(T(\boldsymbol{\kappa}, n, m))$ and k-connectedness can be checked in time $O(T(k,n,m))$. For an undirected graph the bounds are $O(T(\boldsymbol{\kappa}, n, \boldsymbol{\kappa} n))$ and $O(T(k,n,kn))$ respectively. The space is $O(m)$ in all cases.*

Proof of Theorem 2.1 for digraphs and $T = T_1$. We implement the algorithm of Even and Tarjan [7] to compute $\boldsymbol{\kappa}$ as follows. Let x_i, $i = 1, \ldots, n$ be an arbitrary indexing of the vertices.

$i \leftarrow 0$; $\boldsymbol{\kappa} \leftarrow n-1$;
repeat $\{\ i \leftarrow i+1;\ \boldsymbol{\kappa} \leftarrow \min\{\boldsymbol{\kappa},\ \boldsymbol{\kappa}(x_i, G),\ \boldsymbol{\kappa}(x_i, G^R)\}\ \}$ **until** $i > \boldsymbol{\kappa}$;

We refer to this as the *Even-Tarjan algorithm.*

To see this algorithm is correct assume the connectivity is $< n-1$ and let S be a minimum vertex separator of G. When the algorithm halts it has $i > \boldsymbol{\kappa} \geq |S|$. Some x_j does not belong to S for $j \leq |S|+1 \leq i$. Thus $\boldsymbol{\kappa}(x_j) = |S|$ in G or G^R. Since the algorithm computes $\boldsymbol{\kappa}(x_j)$ it halts with $\boldsymbol{\kappa} = |S|$.

It is a simple matter to find a minimum vertex separator assuming each computation of $\boldsymbol{\kappa}(x)$ finds a corresponding separator. To check k-connectedness terminate the loop when $i = k$ and test if $\boldsymbol{\kappa} \geq k$.

The desired time bound with $T = T_1$ follows since each value $\boldsymbol{\kappa}(x)$ is computed in time $O(nm)$.

Proof of Theorem 2.1 for undirected graphs and $T = T_1$. As mentioned, the digraph algorithm applies to an undirected graph. (There is no need to consider G^R.) To check k-connectedness we apply the digraph algorithm to the forest subgraph FG_k of G (this subgraph, constructed by the algorithm of Nagamochi and Ibaraki [17], is defined in Section 1). Since FG_k has $O(kn)$ edges and is found in time $O(m)$ [17] we achieve the desired time bound.

We can also compute the connectivity by working on a subgraph of $O(\boldsymbol{\kappa} n)$ edges. There are two ways to achieve this. The first is to begin by finding a value k with $k \le \boldsymbol{\kappa} < 2k$ (do this by checking k-connectedness for k doubling; alternatively use the algorithm of [12]); then run our connectivity algorithm on the forest subgraph FG_{2k}. We give a second method that may be superior in practice because it is one simple loop. (Readers only interested in asymptotic results can skip this.)

Let $G = (V, E)$ be the given graph. Index the vertices of V as x_i, $i = 1, \ldots, n$. Also let $x_{n+i} = x_i$ and $FG_{n+i} = FG_n = G$ for $i = 1, \ldots, n$.

$i \leftarrow 0$; $\boldsymbol{\kappa} \leftarrow n - 1$;
repeat
  { $i \leftarrow i + 1$; **if** $\boldsymbol{\kappa}(x_i, FG_i) < \min\{i, \boldsymbol{\kappa}\}$ **then** $\boldsymbol{\kappa} \leftarrow \boldsymbol{\kappa}(x_i, FG_i)$ }
until $i > 2\boldsymbol{\kappa}$;

To prove this algorithm is correct let $\boldsymbol{\kappa}^*$ denote the connectivity of G. First observe that the iterations with $i \le \boldsymbol{\kappa}^*$ all have $\boldsymbol{\kappa} = n - 1$. This follows since FG_i is i-connected so $\boldsymbol{\kappa}(x_i, FG_i) \ge i$. When $i > \boldsymbol{\kappa}^*$, FG_i has connectivity $\boldsymbol{\kappa}^*$. Thus $\boldsymbol{\kappa} \ge \boldsymbol{\kappa}^*$ and the algorithm does not halt with $i \le 2\boldsymbol{\kappa}^*$.

Now assume $\boldsymbol{\kappa}^* < n - 1$ and consider the iterations for $i = \boldsymbol{\kappa}^* + 1, \ldots, 2\boldsymbol{\kappa}^* + 1$. Let S be a separating set of G with $\boldsymbol{\kappa}^*$ vertices. S is a separating set of each FG_i. At least one of these $\boldsymbol{\kappa}^* + 1$ iterations has $x_i \notin S$ (note that each iteration examines a distinct x_i). This iteration sets $\boldsymbol{\kappa} = \boldsymbol{\kappa}^*$. Furthermore any corresponding vertex separator of FG_i is a vertex separator of G, since $i > \boldsymbol{\kappa}^*$ (see the discussion of the forest subgraph in Section 1). Thus the algorithm halts with the correct connectivity and separator, and $i = 2\boldsymbol{\kappa}^* + 1$. The latter bound on the number of iterations implies the desired time bound.

Proof of Theorem 2.1 for checking k-connectedness and $T = T_2$. The algorithm is an implementation of a variant of Even's algorithm for checking k-connectedness [6].

Given a digraph G and parameter k, the algorithm either verifies that $\boldsymbol{\kappa} \ge k$ or it finds $\boldsymbol{\kappa}$ and a corresponding separator. Define $\boldsymbol{\kappa}^-$ as $\min\{\boldsymbol{\kappa}, k\}$ and similarly for $\boldsymbol{\kappa}^-(x, y)$. Thus we seek $\boldsymbol{\kappa}^-$ and a corresponding separator if $\boldsymbol{\kappa}^- < k$.

For $X \subseteq V$ and $y \in V - X$, $S \subseteq V - \{y\}$ is a *weak X, y-separator* if $G - S$ contains no path from X to y. Equivalently a weak X, y-separator is a subset of $V - \{y\}$ that is an x, y-separator for every $x \in X$ that it does not contain. Note that X is trivially a weak X, y-separator. Define $\boldsymbol{\kappa}_W(X, y)$ as the smallest cardinality of a weak X, y-separator; also $\boldsymbol{\kappa}_W(X) = \min\{\boldsymbol{\kappa}(X, y) : y \in V - X\}$. If this quantity is less than $|X|$ then it equals the cardinality of an x, y-separator for some $x \in X$.

Choose two arbitrary sets $X, Y \subseteq V$ of cardinality k. (These sets can be equal. We will need the sets to be distinct when we use this algorithm to compute $\boldsymbol{\kappa}$.) Observe that

$$\boldsymbol{\kappa}^- = \min\{\boldsymbol{\kappa}_W(X)\} \cup \{\boldsymbol{\kappa}_W(Y, G^R)\} \cup \{\boldsymbol{\kappa}^-(y, x) : y \in Y, x \in X\}. \tag{1}$$

In proof, each quantity on the right-hand side of (1) corresponds to a separator if it is less than k, so the right-hand side is $\ge \boldsymbol{\kappa}^-$. Next suppose $\boldsymbol{\kappa} < k$ and let S be a minimum separator of $\boldsymbol{\kappa}$ vertices. Thus $V - S$ can be partitioned into two nonempty sets A, B such that $G - S$ contains no path from A to B. If $X \subseteq A \cup S$ then $\boldsymbol{\kappa}_W(X) = |S|$ (we use the fact that $|X| = k$). Similarly if

$Y \subseteq B \cup S$ then $\boldsymbol{\kappa}_W(Y, G^R) = |S|$. In the remaining case X contains a vertex of B and Y contains a vertex of A. Thus $\boldsymbol{\kappa}$ equals the minimum value in the third set of (1).

Our algorithm works by computing the quantities on the right-hand side of (1) and corresponding separators (if such exist). We now describe how each of these quantities is computed.

Consider the first quantity $\boldsymbol{\kappa}_W(X)$. Define a digraph $\widetilde{G}$ by starting with G and adding a new vertex a with edges $ax, x \in X$. Fix a vertex $y \in V - X$. A subset of $V - y$ is an a, y-separator in $\widetilde{G}$ if and only if it is a weak X, y-separator in G. Hence $\boldsymbol{\kappa}_W(X) = \boldsymbol{\kappa}(a, \widetilde{G})$, and we can compute $\boldsymbol{\kappa}_W(X)$ using our procedure to compute $\boldsymbol{\kappa}(x)$. Similarly we compute the second quantity of (1), $\boldsymbol{\kappa}_W(Y, G^R)$.

Finally we compute each value $\boldsymbol{\kappa}^-(y, x)$, $y \in Y, x \in X$. As noted above, $\boldsymbol{\kappa}(y, x)$ equals the size of a minimum $y_S x_T$-cut in the split graph SG. Thus we can compute $\boldsymbol{\kappa}^-(y, x)$ by computing a maximum flow from y_S to x_T, stopping if we verify that the maximum flow value is $\geq k$.

Now we prove the desired time bound $T = T_2$. It suffices to show this for digraphs. The time to compute $\boldsymbol{\kappa}_W(X)$ and $\boldsymbol{\kappa}_W(Y, G^R)$ is $O(nm)$ by our discussion of $\boldsymbol{\kappa}(x)$. We compute each value $\boldsymbol{\kappa}^-(y, x)$ by the Ford-Fulkerson maximum flow algorithm, computing $\leq k$ augmenting paths. Hence the time is $O(km)$. The time to compute all k^2 of these values is $O(k^3 m)$. This establishes the time bound.

Proof of Theorem 2.1 for computing connectivity and $T = T_2$. We find the connectivity $\boldsymbol{\kappa}$ using this algorithm: Check k-connectedness for k equal to successive powers of 2 until we reach a power $2^i > \boldsymbol{\kappa}$. At that point the algorithm determines $\boldsymbol{\kappa}$ and a corresponding separator. (Recall our checking algorithm finds $\boldsymbol{\kappa}$ and a corresponding separator when $k > \boldsymbol{\kappa}$.)

If the graph is undirected it is easy to see that the total time is $O(T_2(\boldsymbol{\kappa}, n, \boldsymbol{\kappa} n))$. Now consider a digraph. The desired bound $O(T_2(\boldsymbol{\kappa}, n, m))$ holds except for the computations of $\boldsymbol{\kappa}_W(X)$ and $\boldsymbol{\kappa}_W(Y, G^R)$. This takes time $O(nm)$ in each k-connectedness check, giving a total contribution of $O(nm \log \boldsymbol{\kappa})$. Now we reduce this to $O(nm)$. Specifically we show that one execution of our VC split algorithm gives enough information to compute each desired value $\boldsymbol{\kappa}_W(X)$ in $O(n)$ time. (Hence the total time is $O(nm + n \log \boldsymbol{\kappa}) = O(nm)$.) The values $\boldsymbol{\kappa}_W(Y, G^R)$ are computed the same way.

The procedure we have given to compute $\boldsymbol{\kappa}_W(X) = \boldsymbol{\kappa}(a, \widetilde{G})$ works as follows: It constructs a contracted graph CG by starting with the split graph of $\widetilde{G}$ and contracting vertices a_S, a_T and x_T for $x \in X$, into a new vertex X_T. Then it finds a minimum X_T-split of the vertex cover $\{y_T : y_T \text{ a vertex of } UG\}$ in the unit capacity graph UG.

Let SG denote the split graph of G. Let $U'G$ be the unit capacity graph formed from SG in a manner similar to UG, i.e., delete all edges $v_S v_T$ and assign capacity 1 to all edges $v_S w_T$. (Note that we have not used a contracted graph CG to form $U'G$.) Choose an arbitrary $z \in V$. Suppose we execute our algorithm to find a minimum z_T-split of vertex cover $C = \{y_T : y \in V\}$ of digraph $U'G$. We shall see that our algorithm works as follows: It orders the vertices of $C - z_T$ as t_i, $i = 1, \ldots, n-1$. (The ordering is determined by the algorithm.) For each such i it computes μ_i as the value of a minimum St_i-cut where $S = \{z_T, t_1, \ldots, t_{i-1}\}$.

Our connectivity algorithm saves these values μ_i. Then for each k-connectedness check, we choose X to be $\{z_T, t_i : 1 \leq i < k\}$. The value $\boldsymbol{\kappa}_W(X) = \boldsymbol{\kappa}(a, \widetilde{G})$ equals $\min\{\mu_i : i = k, \ldots, n-1\}$.

This follows because the contracted graph CG in the computation of $\boldsymbol{\kappa}(a, \widetilde{G})$ is the same as the split graph of G with vertices x_T, $x \in X$ contracted.

Proof of Theorem 2.1 concluded. It is easy to combine the two approaches to connectivity into one algorithm: We execute the second algorithm to find $\boldsymbol{\kappa}$ or determine that $\boldsymbol{\kappa} > \sqrt{n}$. In the latter case we then switch to the first algorithm. This gives one algorithm that achieves Theorem 2.1. $\square$

We mention a related connectivity problem. In a digraph $G = (V, E)$ for any $x \in V$ define

$$\widehat{\boldsymbol{\kappa}}(x) = \min\{\boldsymbol{\kappa}(x, G),\ \boldsymbol{\kappa}(x, G^R)\}.$$

(If G is undirected clearly $\widehat{\boldsymbol{\kappa}} = \boldsymbol{\kappa}$.) The quantity $\widehat{\boldsymbol{\kappa}}(x)$ indicates how well connected x is to the remaining vertices. It may be of interest to know the entire function $\widehat{\boldsymbol{\kappa}}$. For example to choose a "server" in a distributed network we would select a host with the highest connectivity to the "client" nodes, i.e., the server x should maximize $\widehat{\boldsymbol{\kappa}}(x)$.

To calculate all values $\widehat{\boldsymbol{\kappa}}(x)$ first compute $\boldsymbol{\kappa}$ and a corresponding separator S. Observe that any vertex $x \notin S$ has $\widehat{\boldsymbol{\kappa}}(x) = \boldsymbol{\kappa}$. Thus we complete the calculation by computing $\widehat{\boldsymbol{\kappa}}(x)$ for each $x \in S$.

Corollary 2.2 *In a digraph all values $\widehat{\boldsymbol{\kappa}}(x)$, $x \in V$ can be found in time $O(\boldsymbol{\kappa} nm)$.* $\square$

Note that in an undirected graph we can also achieve the time bound $O(\boldsymbol{\kappa}\boldsymbol{\kappa}^* n^2)$ where $\boldsymbol{\kappa}^*$ is the largest value of $\widehat{\boldsymbol{\kappa}}$. For this do the second step using graphs FG_k with k doubling, until all values of $\widehat{\boldsymbol{\kappa}}$ have been found.

In preparation for the next result we refine our time bound for very high connectivities ($\boldsymbol{\kappa} = n - o(n)$). Recall our algorithm for computing $\boldsymbol{\kappa}(x)$. The contracted graph CG contains a vertex y_T only when $xy \notin E$. To state the main observation, for each vertex $x \in V$ define $d_{\mathrm{o}}(x)$ to be the number of edges leaving x (in the given graph G). Then the number of vertices y_T in CG is

$$n - d_{\mathrm{o}}(x). \tag{2}$$

For computing the connectivity note that $d_{\mathrm{o}}(x) \geq \boldsymbol{\kappa}$. Thus (2) implies CG has $\leq n - \boldsymbol{\kappa}$ vertices y_T and $O(n(n - \boldsymbol{\kappa}))$ edges. The same bounds hold when CG is constructed from G^R. Hence the time bound $O(n_T m)$ of Theorem 3.2 implies that we find $\boldsymbol{\kappa}(x)$ in time $O((n-\boldsymbol{\kappa})m) = O(n(n-\boldsymbol{\kappa})^2)$. Thus the Even-Tarjan algorithm given above to compute $\boldsymbol{\kappa}$ runs in time $O(((n-\boldsymbol{\kappa})n)^2)$ (for digraphs or undirected graphs). For example this time bound is linear when $\boldsymbol{\kappa} = n - O(1)$. A similar bound holds for checking k-connectedness.

We turn to randomized algorithms. We compute the connectivity as follows. Define δ as the smallest in-degree or out-degree of a vertex. Our randomized algorithm is the Even-Tarjan algorithm, except that we choose each vertex x_i randomly and terminate when $i \geq 1/\log(n/\delta)$.

To state the results define

$$T(n, m) = nm.$$

Theorem 2.3 *The following can be done with error probability $\leq 1/2$: For a digraph the connectivity $\boldsymbol{\kappa}$ can be found in time $O(T(n, m))$. For an undirected graph $\boldsymbol{\kappa}$ can be found in time*

$O(T(n, \boldsymbol{\kappa} n))$ and k-connectedness can be checked in time $O(T(n, kn))$. For any fixed nonnegative value f, error probability $\leq 1/n^f$ can be achieved in the same time bounds if $\boldsymbol{\kappa}$ or k is $O(n^e)$ for any fixed value $e < 1$. In all cases the space is $O(m)$.

Proof: First consider digraphs. To calculate the error probability of our algorithm let S be a minimum vertex separator. To err the algorithm must choose all i vertices from S. This occurs with probability $\leq (|S|/n)^i \leq (\delta/n)^i \leq 1/2$.

To bound the time write $\delta = (1-\epsilon)n$ for ϵ between 0 and 1. Using natural logarithms, $\ln(n/\delta) = -\ln(1-\epsilon) > \epsilon$. Thus since $1/\epsilon \geq 1$ the number of iterations is $O(1/\epsilon)$. Using (2) and Theorem 3.2, each iteration computes $\boldsymbol{\kappa}(x_i)$ in time $O((n-\delta)m) = O(\epsilon nm)$. Thus the total time is $O(nm)$ as desired.

Now consider undirected graphs. We check k-connectedness using the digraph algorithm on FG_k.

To compute $\boldsymbol{\kappa}$, find a value $k \in (\boldsymbol{\kappa}, 2\boldsymbol{\kappa}]$ and use the digraph algorithm to compute the connectivity of FG_k. We find the desired k using the algorithm of [12] in time $O(\boldsymbol{\kappa} n^2)$ or less. This implies the desired time bound.

For the last part of the theorem first consider digraphs. We change the termination test to $i \geq f/(1-e)$. It is easy to see this achieves the desired error probability. The time bound follows since each iteration uses time $O(nm)$. Undirected graphs are similar. $\square$

To make this paper self-contained we sketch a procedure that for a given undirected graph computes a value $k \in (\boldsymbol{\kappa}, 4\boldsymbol{\kappa})$ in time $O(\boldsymbol{\kappa} n^2)$. This procedure is conceptually simple and can replace that of [12] in the above algorithm to compute undirected connectivity.

Our algorithm is based on the following fact. Let x be a vertex of degree exactly k in FG_k. (The existence of such a vertex is proved in [17]. x can be chosen as the last vertex scanned by the algorithm.) Then $\boldsymbol{\kappa}(x, FG_k)$ equals k if $k \leq \boldsymbol{\kappa}$ and is $< k$ if $k \geq 2\boldsymbol{\kappa}$.

To prove the fact define N as the set of all neighbors of x. If $k \leq \boldsymbol{\kappa}$ then FG_k is k-connected. Thus N is a separator (or trivializor) that shows $\boldsymbol{\kappa}(x, FG_k) = k$.

Suppose $k \geq 2\boldsymbol{\kappa}$. Define S as a vertex separator of $\boldsymbol{\kappa}$ vertices. The desired conclusion $\boldsymbol{\kappa}(x, FG_k) < k$ is clear if $x \notin S$ ($\boldsymbol{\kappa}(x, FG_k) = \boldsymbol{\kappa}$) or if FG_k contains a vertex of degree $< k$. Suppose neither of these is the case. Partition $V - S$ into nonempty sets A and B such that $G - S$ contains no path from A to B. Without loss of generality $|A \cap N| \leq k/2$. Consider the set $T = (S - x) \cup (A \cap N)$. Note that $A - N$ contains some vertex a. (If $A - N = \emptyset$ then $|A| \leq k/2$. This implies any vertex of A has degree $\leq (k/2 - 1) + |S| \leq k - 1 < k$, contradicting our initial assumption.) Thus T is an a,x-separator. Furthermore $|T| \leq (\boldsymbol{\kappa} - 1) + k/2 \leq k - 1$. Thus T shows $\boldsymbol{\kappa}(x, FG_k) < k$ as desired.

The algorithm to compute $k \in (\boldsymbol{\kappa}, 4\boldsymbol{\kappa})$ calculates $\boldsymbol{\kappa}(x, FG_k)$ for k doubling until $\boldsymbol{\kappa}(x, FG_k) < k$. The above fact shows the algorithm halts with k larger than $\boldsymbol{\kappa}$ but no larger than the power of 2 in $[2\boldsymbol{\kappa}, 4\boldsymbol{\kappa})$. The time is $O(\boldsymbol{\kappa} n^2)$ since each calculation of $\boldsymbol{\kappa}(x)$ uses time $O(kn^2)$.

The final topic of this section is computing weighted vertex connectivity. Let $w : V \to \mathbf{R}_+$ be a vertex weight function. Our algorithm first finds a minimum unweighted separator S (using one of the previous algorithms). Then it sets

$$\boldsymbol{\kappa} = \min\{w(S),\ \boldsymbol{\kappa}(x,G),\ \boldsymbol{\kappa}(x,G^R) : x \in S\}.$$

(All references to $\boldsymbol{\kappa}$ refer to weighted connectivity.) This is correct since if S is not a minimum weight separator then some $x \in S$ is not contained in some minimum weight separator, so the weighted connectivity is $\boldsymbol{\kappa}(x,G)$ or $\boldsymbol{\kappa}(x,G^R)$.

To calculate the values $\boldsymbol{\kappa}(x,G)$ and $\boldsymbol{\kappa}(x,G^R)$ we proceed similar to the unweighted case: Define the split graph SG as before, except that each edge $v_T v_S$ has capacity $w(v)$. Define the contracted graph CG as before. Applying Theorem 3.3 to CG shows $\boldsymbol{\kappa}(x,G)$ can be found in time $O(nm \log(n^2/m))$.

Theorem 2.4 *For any digraph or undirected graph with nonnegative vertex weights, the weighted connectivity can be found in time $O(\boldsymbol{\kappa}_1 nm \log(n^2/m))$, where $\boldsymbol{\kappa}_1$ is the unweighted vertex connectivity. The space is $O(m)$.* □

A randomized algorithm also works. Define

$$\delta = \min\{w(\{x : xv \in E\}),\ w(\{x : vx \in E\}) : v \in V\}.$$

We use the Even-Tarjan algorithm, except that we choose each vertex x_i randomly with any $v \in V$ having probability $w(v)/w(V)$, and we terminate when $i \ge 1/\log(w(V)/\delta)$.

Theorem 2.5 *For any digraph or undirected graph with nonnegative vertex weights, the weighted connectivity can be found with error probability $\le 1/2$ in expected time $O(nm \log(n^2/m))$. The space is $O(m)$.*

Proof: To calculate the error probability let S be a minimum weight vertex separator. To err the algorithm must choose all i vertices from S. This occurs with probability $\le (\boldsymbol{\kappa}/w(V))^i$. Note that $\delta \ge \boldsymbol{\kappa}$, since all quantities in the definition of δ are the weights of vertex separators or trivializors. Thus the error probability is $\le (\delta/w(V))^i \le 1/2$.

To bound the expected time write $\delta = (1-\epsilon)w(V)$ for ϵ between 0 and 1. The same calculation as Theorem 2.3 shows there are $O(1/\epsilon)$ iterations. Thus it suffices to show the expected time for an iteration is $O(\epsilon nm \log(n^2/m))$.

Equation (2) gives the size of our vertex cover of CG (recall property (iii) of CG). Hence we can apply Theorem 3.3 with $n_T = n - d_o(x)$. It shows that any quantity $\boldsymbol{\kappa}(x,G)$ is computed in time $O((n - d_o(x))m \log(n^2/m))$. The expected value of $d_o(x)$ is

$$\sum\{w(x)d_o(x)/w(V) : x \in V\} = (1/w(V))\sum\{w(\{x : xv \in E\}) : v \in V\} \ge n\delta/w(V),$$

since any vertex v has $w(\{x : xv \in E\}) \ge \delta$. Hence $n - d_o(x)$ has expectation $\le n(1-\delta/w(V)) = n\epsilon$. So the expected time to compute $\boldsymbol{\kappa}(x,G)$ is $O(\epsilon nm \log(n^2/m))$. Similarly for computing $\boldsymbol{\kappa}(x,G^R)$. Thus the expected time for an iteration is as desired. □

3 Minimum Vertex Cover Splits

This section presents an algorithm to find a minimum vertex cover split. Recall that we are given a digraph with vertex cover T and vertex $s \in T$; we seek an $\overline{s}t$-set that has smallest possible in-degree subject to the constraint that $t \in T$. Our algorithm runs in time $O(nm)$ on a unit capacity multigraph and time $O(nm \log(n^2/m))$ on a capacitated graph. More precisely write $n_T = |T|$. The time bounds are actually $O(n_T m)$ and $O(n_T m \log(2 + n_T^2/m))$ respectively (see Theorems 3.2 and 3.3).

We solve the special case of the problem where the undirected version of G is bipartite and the vertex cover T is one set of the bipartition. The general problem reduces to this special case as follows. Let an instance of the general problem be specified by digraph $G = (V, E)$, vertex cover T and $s \in T$. Replace each edge $vw \in E$ that has $v, w \in T$ by two edges vx, xw. Here x is a new vertex (on the two new edges and no others) and the two new edges have the same capacity as the original. The new graph is bipartite (no edge joins two vertices of $V - T$ since T is a vertex cover of G). For any $t \in T$ a minimum st-cut has the same value in both graphs (since a maximum st-flow has the same value). The parameters n_T and m change by at most a constant factor. Hence it suffices to consider the bipartite case. (We note that a version of our split algorithm works directly on the general problem. We do not present this version because the `Relabel` routine has more cases. We also note that the graphs generated by the reductions of Section 2 are already bipartite.)

We briefly review the preflow-push algorithm (see [10] for a complete treatment). Fix a digraph $G = (V, E)$ with distinguished vertices s (the *source*) and t (the *sink*), and capacity function $c : E \to \mathbf{R}_+$. Without loss of generality assume an edge e belongs to E if and only if its reverse edge, denoted e^R, belongs to E. A *flow* is function $f : E \to \mathbf{R}_+$ satisfying the *capacity constraint* (for each edge $e \in E$, $f(e) \le c(e)$) and the *conservation constraint* (each vertex $\ne s, t$ has excess 0, where the *excess* at vertex v is $e(v) = f(\{wv : wv \in E\}) - f(\{vw : vw \in E\})$). A *preflow* obeys the same conditions except that instead of conservation we require that each vertex $v \ne s$ has $e(v) \ge 0$. The *value* of a flow or preflow is $e(t)$; a *maximum flow (preflow)* is one with maximum value. The maximum value of a preflow is the same as that of a flow, because any preflow can be converted to a flow of the same value.

Fix a preflow f. A vertex $v \ne s, t$ is *overflowing* if $e(v) > 0$. An edge e has *residual capacity* $r(e) = c(e) - f(e) + f(e^R)$. An edge with $r(e) > 0$ is a *residual edge*; an edge with $r(e) = 0$ is *saturated*. To *push* δ units of flow along e means to increase $f(e)$ by δ, and then reduce both $f(e)$ and $f(e^R)$ by $\min\{f(e), f(e^R)\}$. Pushing up to $\min\{e(v), r(vw)\}$ units of flow along edge vw keeps f a valid preflow. A push of $r(vw)$ units is a *saturating push* (it makes vw saturated).

A *distance function* is a mapping $d : V \to \mathbf{N}$ such that any residual edge vw not incident to s satisfies $d(v) \le d(w) + 1$.

The preflow-push algorithm of [10] computes a maximum preflow. To do this it maintains a preflow and corresponding distance function, repeatedly pushing flow on residual edges vw that have $d(v) = d(w) + 1$, and increasing distance labels, until there are no overflowing vertices. At that point the preflow is maximum and the algorithm halts.

The algorithm of [13] finds a minimum unrestricted cut (i.e., an xy-cut of minimum capacity where x and y are arbitrary). It maintains a preflow from source S to sink t. Here S is a contraction of vertices of V (we use S to denote both the subset of V and its contraction); t is a vertex not in S. Initially S consists of one vertex $s \in V$. The algorithm repeatedly finds a maximum preflow from S to t (following [10]) and then transfers t to S and chooses a new sink vertex $t' \notin S$ that has minimum distance $d(t')$. At certain points the algorithm makes a set of vertices "dormant." The dormant vertices are temporarily ignored – the algorithm pushes flow and increases distance labels only on vertices that are "awake" (i.e., vertices not belonging to S or a dormant set). When no awake vertices remain (because they have all been chosen as sinks and transferred to S) the algorithm awakens the last dormant set and repeats this process on them. As mentioned, for each sink t the algorithm computes a maximum preflow from S to t; its value equals that of a minimum St-cut. The minimum unrestricted cut of the given graph G has value equal to the smallest of these St-cuts on G or G^R.

Our algorithm extends the Hao-Orlin minimum unrestricted cut algorithm. The main change is a new operation `Join` which adjoins an awake vertex to a previously-created dormant set.

3.1 Algorithm Statement

We present the algorithm using the same organization as [13]: We divide the algorithm into a number of subroutines. We state each routine and then briefly comment on how it works and how it differs from [13]. The formal analysis of our algorithm does not rely on these comments.

We use the following notation. The algorithm maintains a partition of V into sets S (the source set), D_i, $i = 1, \ldots, \gamma$ (the dormant sets) and W (the awake set). Let $\mathcal{P}$ denote this partition. ($\mathcal{P}$ changes over time.) Also define $D_0 \equiv S$. The function d is the distance function. For a set $U \subseteq V$, $d_{\min}(U) = \min\{d(v) : v \in U\}$ and $d_{\max}(U) = \max\{d(v) : v \in U\}$.

Recall that we are given G, T, s where G is a digraph whose undirected version is bipartite, T is one set of the bipartition and $s \in T$. We seek an $\overline{s}$-set that contains a vertex of T and has smallest possible in-degree. The following main routine returns with W^* equal to the desired set.

procedure `Min_VC_Split`(G, T, s);
`Initialize`; /* initialize flow, distances $d(v)$, etc. */
repeat
  { `New_Sink`; /* choose new sink, etc. */
    while a vertex of W is overflowing **do**
      { choose an overflowing vertex $v \in W$;
        if there is a residual edge vw with $w \in W$ and $d(v) = d(w) + 1$ **then**
          push $\min\{e(v), r(vw)\}$ units of flow along vw
        else `Relabel`(v) }
    if $\rho(W, G) < \rho(W^*, G)$ **then** $W^* \leftarrow W$ }
until $T \subseteq S \cup t$;

This is essentially the main routine of [13]. The `Initialize` and `New_Sink` routines collectively initialize the data structures for the first iteration. Here is `Initialize`; `New_Sink`, which selects the next sink t of the preflow, is given later.

procedure `Initialize`;

1. $W^* \leftarrow$ any $\overline{s}$-set separating T, e.g., $\{v\}$ for some $v \in T - s$;
2. $S \leftarrow \emptyset$; $\gamma \leftarrow 0$; /* S is the same set as D_0 */
3. $W \leftarrow V$; $t \leftarrow s$;
4. set the flow in each edge of G to 0;
5. **for** $v \in V$ **do if** $v \in T$ **then** $d(v) \leftarrow 0$ **else** $d(v) \leftarrow 1$;

We shall see that in the first iteration `New_Sink` completes the initialization by setting S to $\{s\}$ and t to the first sink, and saturating the edges directed from s.

The `Relabel` routine either increases the distance label $d(v)$ or adjusts the dormant sets. The `Create_Dormant` routine creates a new dormant set as in [13]. The `Join` routine adjoins v to an existing dormant set.

procedure `Relabel`(v); /* $v \in W$ */

1. **if** v is the only vertex in W at distance $d(v)$ and $d(v) < d_{\max}(W)$ **then**
2.   `Create_Dormant`$(\{u \in W : d(u) \geq d(v)\})$
3. **else if** no residual edge goes from v to W **then**
4.   **if** $v \in T$ **then** `Create_Dormant`$(\{v\})$ **else** `Join`(v)
5. **else** $d(v) \leftarrow \min\{d(w)+1 : vw \text{ a residual edge}, w \in W\}$;

procedure `Create_Dormant`(U); /* $U \subseteq W$ */

1. $W \leftarrow W - U$; $\gamma \leftarrow \gamma + 1$; $D_\gamma \leftarrow U$;

procedure `Join`(v); /* $v \in W - T$ */

1. $W \leftarrow W - v$; $D_\gamma \leftarrow D_\gamma \cup v$; $d(v) \leftarrow d_{\min}(D_\gamma \cap T) + 1$;

In line 1 of `Relabel` if v is the only awake vertex at distance $d(v)$ and $d(v) = d_{\max}(W)$ then the condition of line 3 holds, i.e., no residual edge goes from v to W. (This follows from the d-Validity Property below, which says d is a valid distance function.) In `Join` the resetting of $d(v)$ preserves the validity of d. This resetting can change $d(v)$ arbitrarily, in particular $d(v)$ can decrease. This contrasts with most preflow-push algorithms where any vertex v has $d(v)$ nondecreasing (e.g., [1, 2, 10, 11, 13, 14]). Finally note that `Join` can add $v \notin T$ to $S = D_0$.

`New_Sink` transfers the previous sink to S and saturates all edges directed from it. Then it chooses an awake vertex of T as the next sink. This may necessitate waking the last dormant set D_γ.

procedure `New_Sink`;

1. $W \leftarrow W - t$; $S \leftarrow S \cup t$;
2. saturate each residual edge tv directed from t;
3. **if** $W \cap T = \emptyset$ **then**
4.   { **for** $v \in W$ **do** `Join`(v);
5. $W \leftarrow D_\gamma$; $\gamma \leftarrow \gamma - 1$ }
6. **for** each $v \in W - T$ with $d(v) = d_{\min}(W)$ **do** $d(v) \leftarrow d(v) + 2$;
7. choose t as a vertex of W with $d(t) = d_{\min}(W)$;

The purpose of the `Join` operations in line 4 is to update distance labels. Now consider the loop of line 6 that increases distance values by 2. It applies when the awake vertex $t' \in T$ with smallest distance value has $d(t') = d_{\min}(W)+1$. Thus the loop ensures that the new sink t' has the smallest distance value. This is needed in the following two situations.

In both situations, `New_Sink` is entered with the old sink t being the only vertex at distance $d(t)$. The first situation is when some other vertex of T is awake. We shall see this implies there is such a vertex t' with $d(t') = d(t) + 2$. This quantity equals $d_{\min}(W) + 1$ after line 1 deletes t from W. Hence the loop ensures $d(t') = d_{\min}(W)$ as desired. The second situation is when no other vertex of T is awake, and line 5 wakens D_γ. If D_γ was created (in `Relabel`) when v was the only awake vertex at distance $d(v)$ for some $v \notin T$, then a vertex $t' \in T \cap D_\gamma$ with smallest distance value has $d(t') = d(v) + 1$. Thus again the loop ensures $d(t') = d_{\min}(W)$. This concludes the statement of our algorithm.

3.2 Algorithm Analysis

We prove that our algorithm has properties similar to the ones shown in [13]. We state these nine properties below and discuss how they are used in the overall argument. Then we present the proofs of the properties.

The first two properties, besides being crucial overall, will ensure that the algorithm is well-defined and halts. Recall the definition of $\mathcal{P}$ from the start of Section 3.1.

Parity Property. *$d(v)$ is even if $v \in T$ and odd if $v \notin T$.*

d-Consecutiveness Property. *If $U \in \mathcal{P} - S$ then $d(U)$ is a set of consecutive integers.*

The next paragraph shows the above two properties imply that a dormant set always contains a vertex of T. This fact has two important consequences. First it makes the algorithm well-defined – in `Join`, the assignment to $d(v)$ needs $D_\gamma \cap T \neq \emptyset$. Second it guarantees that line 7 of `New_Sink` always chooses a sink t that belongs to T. (This follows by examining lines 3–7 of `New_Sink` and using the fact, d-Consecutiveness and Parity.)

We now prove that a dormant set always contains a vertex of T. The dormant set D_0 contains $s \in T$ (after the first execution of `New_Sink`). A dormant set created in line 4 of `Relabel` (by `Create_Dormant`) clearly contains a vertex of T. Consider a dormant set created in line 2 of `Relabel`. The new dormant set has vertices with at least two distinct distance values. Hence the desired property follows from d-Consecutiveness and Parity.

The next two properties are important for correctness of the algorithm.

d-Validity Property. *A residual edge vw with v and w in the same set of $\mathcal{P}-S$ has $d(v) \leq d(w)+1$.*

Dormancy Property. *No residual edge enters W or goes from D_i to D_j, $0 \leq i < j$.*

The next property implies the algorithm is correct.

Optimality Property. *Whenever* `Min_VC_Split` *updates W^*, W is an $\overline{S \cap T}$ t-set of minimum in-degree $\rho(W, G)$.*

To see this implies correctness, let U be a minimum s-split of T. Let t be the first vertex of U chosen as the sink by `New_Sink`. Thus when t is the sink U is an $\overline{S \cap T}\,t$-set. (Note that U need not be an $\overline{S}t$-set, i.e., U may contain vertices of $S - T$.) Now the Optimality Property implies `Min_VC_Split` finds a set W of the same in-degree as U.

The last four properties are used in the efficiency analysis.

Increasing Distance Property. *A value $d(v)$ can decrease only in an operation* `Join`(v). *An operation* `Relabel`(v) *either increases $d(v)$ or makes v dormant.*

The main purpose of the next property is to bound the total increase in $d(v)$ in operations other than `Join`(v). For any vertex $v \in V$ define the following measure of increase:

$$
\begin{aligned}
I(v) = {} & (\text{the total amount that } d(v) \text{ has increased in all operations other than } \texttt{Join}(v)) \\
& +2 \times (\text{the number of operations } \texttt{Join}(v)).
\end{aligned}
$$

A vertex $v \in T$ is never in a join operation. Hence at any moment $I(v) = d(v)$ for any $v \in T$.

d-Bound Property. *Each value $d(v)$ is less than $2n_T$. Each value $I(v)$ is at most $2n_T$.*

Transfer Bound Property. `New_Sink` *awakens fewer than n_T dormant sets.* `Create_Dormant` *creates fewer than n_T dormant sets. A given vertex becomes dormant fewer than n_T times.*

Saturation Bound Property. *Any edge and its reverse are collectively saturated at most $2n_T$ times.*

Now we prove the properties. The arguments are similar to [13] except for the d-Bound. For the first property and others note that the algorithm changes distance values in four places: `Initialize` (line 5), `Relabel` (line 5), `Join`, and `New_Sink` (line 6).

Proof of Parity. This property is first established by `Initialize`. It is preserved when line 5 of `Relabel` changes a distance value because G is bipartite. It is obviously preserved when `Join` or line 6 of `New_Sink` changes a distance value. □

Proof of d-Consecutiveness. The argument is by induction on the number of steps of the algorithm. d-Consecutiveness is first established by `Initialize`.

Consider the `Relabel` routine. The call to `Create_Dormant` in line 2 obviously preserves d-Consecutiveness (for both W and the new dormant set). If this call is not made then either $d(v) = d_{\max}(W)$ or there is another vertex at distance $d(v)$. In either case $d(W - v)$ is a set of consecutive integers. This implies the d-Consecutiveness of W is preserved in all other cases of `Relabel` (i.e., in the calls to `Create_Dormant` or `Join` in line 4 and when line 5 redefines $d(v)$). In line 4 d-Consecutiveness is obvious for the new dormant set $\{v\}$ formed by `Create_Dormant`. Finally any call to `Join` preserves d-Consecutiveness for D_γ when it redefines $d(v)$.

Next consider the `New_Sink` routine. d-Consecutiveness is preserved when line 1 deletes t from W since $d(t) = d_{\min}(W)$. The calls to `Join` (line 4) preserve d-Consecutiveness as noted above. Line 6 preserves d-Consecutiveness of W by d-Consecutiveness prior to this line and Parity. □

Proof of d-Validity and Dormancy. These two properties are proved together by induction on the number of steps of the algorithm. Both properties are established when `Initialize` sets the distance values in line 5.

Consider a push operation in line 7 of `Min_VC_Split`. The only residual edge that can be created by the push is wv. d-Validity is preserved since $d(w) < d(v)$ by line 6. Dormancy is preserved since $w \in W$ by line 6.

Consider the `Relabel` routine. We begin by checking Dormancy is preserved. Consider the call to `Create_Dormant` in line 2, and suppose it makes D_γ the new dormant set. Take any residual edge uw with $u \in D_\gamma$, and w belonging to W before the call. Then $d(w) \geq d(v)$, by d-Validity if $u \neq v$ and by line 6 of `Min_VC_Split` if $u = v$. Thus $w \in D_\gamma$ and Dormancy is preserved. Line 4 of `Relabel` preserves Dormancy by the test of line 3.

Next we check that d-Validity is preserved in `Relabel`. The first case to consider is an operation `Join`(v). Take any vertex $w \in D_\gamma$. A residual edge joining v and w is directed from v to w (by Dormancy before `Join`(v)). Also $w \in T$ (since $v \notin T$). Hence the new value of $d(v)$ satisfies the validity inequality.

The second case to consider is when line 5 of `Relabel` redefines $d(v)$. The new value obviously preserves d-Validity for edges directed from v. For edges directed to v observe that this operation increases $d(v)$ (by d-Validity prior to line 5, and the definition of v in line 6 of `Min_VC_Split`).

Finally consider the `New_Sink` routine. Dormancy is preserved when line 2 saturates the edges directed from t. When $W \cap T = \emptyset$ a `Join` operation preserves d-Validity, by the argument above. Also it is obvious that in line 5 redefining W to D_γ preserves Dormancy. Lastly we show line 6 preserves d-Validity. Immediately before line 6 increases a value $d(v)$, any vertex $w \in W \cap T$ has $d(w) \geq d(v) + 1$ by Parity. This implies d-Validity. □

Proof of Optimality. When `Min_VC_Split` updates W^* all vertices with negative excess are in $S \cap T$. Hence we have a preflow from $S \cap T$ to t. The set W contains t but no vertex of $S \cap T$. No vertex of W is overflowing, by definition. Every edge of G entering W is saturated, by Dormancy. This implies we have a maximum value preflow from $S \cap T$ to t and W is an $\overline{S \cap T}\,t$-set of minimum in-degree. □

Proof of Increasing Distance. Line 5 of `Relabel` increases a distance value $d(v)$ (as shown in the proof of d-Validity and Dormancy). This implies the second part of the Increasing Distance Property. Line 6 of `New_Sink` also increases distance values. □

We prove the d-Bound using the following lemma.

Lemma 3.1 *Whenever a vertex v is awake,*

$$I(v) < 2|S \cap T| + d(v) - d(t). \tag{3}$$

Immediately before an operation `Join`(v)*,*

$$I(v) < 2|(S \cup W) \cap T|. \tag{4}$$

Let us show that the lemma implies the d-Bound. Inequality (3) implies the first part of the d-Bound: First consider a vertex $v \in T$. When v is chosen as the sink $d(v)$ has achieved its maximum value and (3) gives $d(v) = I(v) < 2|S \cap T| \le 2(n_T - 1)$. Next consider a vertex $v \notin T$. The algorithm always defines $d(v)$ as one more than the distance of a vertex in T, so we always have $d(v) < 2n_T$.

Inequality (4) essentially implies the second part of the d-Bound. For the second part we need only consider vertices $v \notin T$. First consider a vertex $v \notin T$ that becomes dormant in an operation `Join`(v) and is never subsequently reawakened. Before this join inequality (4) shows $I(v) < 2n_T$. The join increases $I(v)$ by 2, so its final value is $\le 2n_T$ (since $I(v)$ is even). Thus the second part of the d-Bound holds for v. To extend this to all vertices $v \notin T$ we make the convention that when $T \subseteq S \cup t$ in line 10 of `Min_VC_Split`, the algorithm actually continues to execute `New_Sink`, line 4 of which performs `Join`(v) for each v remaining in W. With this convention the previous argument applies to every $v \notin T$. We conclude that Lemma 3.1 implies the d-Bound.

Before proving Lemma 3.1 we give a useful inequality. Consider the moment when an execution of `Create_Dormant`(U) is completed. Let $v \in U \cap T$ have minimum distance, i.e., $d(v) = d_{\min}(U \cap T)$. Then

$$d(v) \le d(t) + 2|W \cap T|. \tag{5}$$

This follows from d-Consecutiveness and Parity. Similar reasoning shows that (5) holds (with strict inequality) immediately before an execution of `Join`(v) in `Relabel`(v).

Proof of Lemma 3.1. The argument is by induction on the number of steps of the algorithm. At the start of the algorithm (after the first execution of `New_Sink`) for any vertex v, $I(v) = d(t) = 0$, $S = \{s\}$ and $d(v) \ge 0$, so (3) holds.

Consider the `Relabel` routine. When line 5 of `Relabel` increases $d(v)$, obviously (3) is preserved. The only remaining case to check in `Relabel` is when `Join`(v) is executed. We wish to show (4) holds right before the join. Inequality (5) shows right before the join,

$$0 \le 2|W \cap T| + d(t) - d(v).$$

Also (3) holds right before the join since v is awake. Adding these inequalities gives (4).

Now consider the `New_Sink` routine. The following three cases exhaust all the possibilities.

Case 1: v is awake throughout `New_Sink`*.* We first claim that no set gets awakened in `New_Sink`. To prove this it suffices to show $W \cap T \neq \emptyset$ in line 3. This holds if $v \in T$ since then $v \in W \cap T$ by Case 1. It holds if $v \notin T$ since `Join`(v) is not executed in Case 1.

Now we show that `New_Sink` does not decrease the quantity $2|S \cap T| - d(t)$ on the right-hand side of (3). Line 1 increases $2|S \cap T|$ by 2. When line 7 chooses a new value for t the quantity $d(t)$ increases by at most 2 from its previous value, by the claim, d-Consecutiveness and Parity. The net effect is that $2|S \cap T| - d(t)$ does not decrease.

Finally if line 6 increases $d(v)$ both sides of (3) increase by 2. We conclude that (3) is preserved in Case 1.

Case 2: v is awake when `New_Sink` *starts and becomes dormant in* `New_Sink`. If $v \in T$ it becomes dormant in line 1 and there is nothing to prove. Suppose $v \notin T$, so v becomes dormant in line

4 when `Join`(v) is executed and v is reawakened in line 5. Immediately before `New_Sink` starts, $d(v) = d(t) + 1$ (by the test of line 3). Thus (3) gives

$$I(v) < 2|S \cap T| + 1. \tag{6}$$

(6) becomes $I(v) < 2|S \cap T| - 1$ when line 1 adds t to S. This gives the desired inequality (2) right before the operation `Join`(v) (note $W \cap T = \emptyset$ at this moment). The join increases $I(v)$ by 2, so (6) holds once again. The join also sets $d(v)$ so that when v awakens in line 5 and the next sink t is chosen in line 7, we have $d(v) = d(t) + 1$. Thus (6) implies that (3) holds at the end of `New_Sink`, as desired.

Case 3: v is dormant when `New_Sink` *starts.* If v remains dormant throughout the execution of `New_Sink` there is nothing to prove. So assume v is awakened in line 5. We examine the two possibilities for how v became dormant.

The first possibility is that an operation `Create_Dormant`(U) made v dormant. Right after the execution of `Create_Dormant`(U) inequality (5) holds, so

$$0 \le 2|W \cap T| + d(t) - d_{\min}(U \cap T).$$

Inequality (3) holds right before `Create_Dormant`(U). It still holds right after `Create_Dormant`(U) since no term changes. Adding (3) to the above inequality gives

$$I(v) < 2|(S \cup W) \cap T| + d(v) - d_{\min}(U \cap T). \tag{7}$$

By the time v gets awakened in `New_Sink`, all vertices in $W \cap T$ in (7) have been transferred to S. Furthermore some vertex at distance $d_{\min}(U \cap T)$ becomes the sink. Hence (7) implies that (3) holds at the end of `New_Sink`, as desired.

The second possibility is that an operation `Join`(v) in `Relabel` made v dormant. Right before `Join`(v) inequality (4) holds. By the time v gets awakened in `New_Sink`, all vertices in $W \cap T$ in this inequality have been transferred to S. Also the `Join`(v) operation increases $I(v)$ by 2. Thus the inequality becomes $I(v) < 2|S \cap T| + 2$. Since $I(v)$ is even this implies $I(v) < 2|S \cap T| + 1$. The join operation sets $d(v)$ so $d(v) = d(t) + 1$ when the next sink t is chosen. This implies that (3) holds at the end of `New_Sink`, as desired. □

Proof of Transfer Bound. There are $n_T - 1$ executions of `New_Sink`, since each one chooses a new vertex of T as sink. This gives the first part of the Transfer Bound. There are $< n_T - 1$ executions of `Create_Dormant`, since each one creates a dormant set which eventually gets awakened in line 5 of `New_Sink`. This gives the second part. A given vertex becomes dormant $< n_T$ times, since each time its new dormant set either equals D_0 or was created by `Create_Dormant`. □

Proof of Saturation Bound. Consider the saturating pushes from v to w or w to v for two fixed vertices $v \notin T$ and $w \in T$. We claim that during the interval of time after one such push and before the next, the quantity $I(v) + d(w)$ increases by at least 2. When the algorithm ends this quantity is $< 4n_T$ (d-Bound). Thus the claim implies there are $\le 2n_T$ saturating pushes total.

Now we prove the claim. The claim is obvious if `Join`(v) occurs in the interval, so assume it does not. Thus neither $d(v)$ nor $d(w)$ decreases (Increasing Distance). Without loss of generality suppose the first saturating push is from v to w. The next push along vw or wv goes from w to v, which implies $d(w)$ has increased by ≥ 2 (line 6 of `Min_VC_Split`). □

Now we estimate the time for the algorithm. We use the inequality $n \leq m$, which holds since a vertex not on any edge can be deleted.

First we give some necessary data structures. We use the current edge data structure to scan the edges incident to a vertex [10]. We use the additional rule that every operation `Join`(v) resets v's current edge pointer to the beginning of its adjacency list.

These rules ensure that whenever v's current edge pointer reaches the end of v's adjacency list, every edge vw has been scanned with $d(v)$ equal to its current value. Hence as in [10] each time v's entire adjacency list has been scanned, `Relabel`(v) is executed. The latter occurs $O(n_T)$ times by the Increasing Distance Property, d-Bound and Transfer Bound. A vertex $v \notin T$ may also have $O(n_T)$ partial scans of its adjacency list, each ending in the operation `Join`(v). We conclude that any given edge is scanned $O(n_T)$ times. Thus the total time for scanning edges (line 6 of `Min_VC_Split` and lines 3 and 5 of `Relabel`) is $O(n_T m)$.

We maintain several lists of awake vertices. To choose an overflowing vertex (in lines 4–5 of `Min_VC_Split`) we maintain a list of the overflowing vertices of W. To implement lines 1–2 of `Relabel` we maintain a list of vertices of W at each distance value between $d_{\min}(W)$ and $d_{\max}(W)$. We also mark each awake vertex, to implement line 6 of `Min_VC_Split` and lines 3 and 5 of `Relabel`. These data structures are initialized each time a dormant set is awakened, in total time $O(n_T(m+n)) = O(n_T m)$.

We record the value $d_{\min}(D \cap T)$ for each dormant set D. Thus each join operation uses time $O(1)$, giving time $O(n_T n)$ total for joins.

Each push takes $O(1)$ time. The Saturation Bound implies there are $O(n_T m)$ saturating pushes. It follows from this discussion that excluding nonsaturating pushes, the total time for `Min_VC_Split` is $O(n_T m)$.

Now suppose `Min_VC_Split` is executed on a unit capacity digraph G. (In such a graph every edge has capacity one but parallel edges are allowed.) This is the case needed for all results of Section 2 except weighted connectivity. For such graphs we can consider the residual graph also to be unit capacity. That is instead of defining the residual capacity of e as $r(e)$, the residual graph contains $r(e)$ copies of e, each having capacity 1. (Thus each edge of G corresponds to a unique edge in any residual graph.) The Saturation Bound still holds. Furthermore in such residual graphs every push is saturating, i.e., there are no nonsaturating pushes. We have thus completely analyzed the time for such graphs and can conclude the following:

Theorem 3.2 *A minimum s-split of a vertex cover T can be found in time $O(n_T m)$ in a digraph with unit edge capacities, for $n_T = |T| \leq n$. The space is $O(m)$.* □

We turn to the implementation of `Min_VC_Split` on capacitated digraphs. We first achieve time $O(n_T m \log(n^2/m))$ by incorporating dynamic trees, and then we improve the logarithmic factor. The modifications and analysis are similar to the maximum flow algorithm of [10]. We begin by summarizing that implementation in the context of `Min_VC_Split`. Then we sketch the changes to our algorithm from [10], and the changes in the analysis of our algorithm. We conclude by sketching how to improve the time bound to $O(n_T m \log(n_T^2/m))$. We assume the reader is familiar with [10].

The dynamic tree algorithm of [10] maintains a forest of dynamic trees. Consider a dynamic tree edge vw. vw is the current edge of v, with w the parent of v in the dynamic tree. Furthermore edge vw satisfies the condition of line 6 of `Min_VC_Split`, so line 7 can push flow along it. The algorithm pushes flow along paths in dynamic trees, specifically along paths from a vertex v to the root of its dynamic tree. The algorithm also pushes flow along single edges not in a dynamic tree. It maintains the following *overflowing vertex invariant*: Any overflowing vertex is the root of the dynamic tree containing it. This is done by using a routine *send* to push flow in a dynamic trees. *send*(v) repeatedly pushes flow from v to the root of its dynamic tree until either v has no excess or v becomes a dynamic tree root. *send* also cuts any newly saturated edge, to preserve the definition of dynamic tree edges. Line 7 of `Min_VC_Split` either links the dynamic trees of v and w and then does *send*(v), or pushes flow along edge vw and then does *send*(w).

When a vertex v is relabelled, i.e., the distance value $d(v)$ is increased as in line 5 of `Relabel`, the algorithm performs an operation *cut-children*(v), where *cut-children*(v) cuts every dynamic tree edge corresponding to a child of v. This preserves the definition of dynamic tree edges.

The algorithm chooses overflowing vertices to process (line 5 of `Min_VC_Split`) using a queue of overflowing vertices (the "FIFO preflow-push algorithm"). Each time an overflowing vertex v is chosen from the head of the queue, pushes from v are done (as described above for line 7 of `Min_VC_Split`) until either the excess of v becomes 0 or a relabel operation is done to increase $d(v)$. A vertex is added to the end of the queue when it becomes overflowing as a result of a push; also if v is relabelled it is added to the end of the queue.

Each dynamic tree is maintained to contain at most k vertices for some parameter value k. Thus each dynamic tree operation uses time $O(\log k)$. Choosing $k = \log(n^2/m)$ gives the time bound $O(nm \log(n^2/m))$ to find a maximum value flow [10].

The dynamic tree implementation of `Min_VC_Split` is the same as [10]. It does some additional operations that keep each dynamic tree contained in one set of the partition $\mathcal{P}$, and other additional operations corresponding to the extra operations of `Min_VC_Split` compared to [10]. The additional operations are as follows.

Consider `Relabel`(v). Recall that v is the root of its dynamic tree since it is overflowing. If the condition of line 3 of `Relabel` holds then *cut-children*(v) is done before `Create_Dormant`$(\{v\})$ or `Join`(v) is called in line 4. The call to `Create_Dormant` in line 2 transfers every dynamic tree of W whose root is at distance $\geq d(v)$ to the new dormant set D_γ. (D_γ equals the set of all vertices in these trees. In proof, v's dynamic tree is the only dynamic tree of W that contains a vertex at distance $d(v)$, by the test of line 1. This implies any dynamic tree containing a vertex of D_γ has its root at distance $\geq d(v)$, since each edge uw of a dynamic tree has $d(u) = d(w) + 1$ and the root has the smallest distance value.)

Next consider `New_Sink`. Line 1 does *cut-children*(t) when t is deleted from W. Line 6 does *cut-children*(v) when $d(v)$ is increased by 2.

Lastly note that at the end of `New_Sink` the overflowing vertex invariant can be violated: A vertex $w \in W$ can have positive excess without being a dynamic tree root. (This is caused by the edge saturations in line 2 of `New_Sink`. These saturations could occur in previous executions of `New_Sink` if W was just awakened in line 5.) To restore the invariant `New_Sink` performs *send*(w) for each such w.

We turn to the analysis of the dynamic tree algorithm. Cut-children operations do $O(n_T m)$ extra dynamic tree cut operations. This follows since after an operation *cut-children*(v), v becomes dormant or $d(v)$ increases. `New_Sink` does $\leq m$ new *send* operations. Since this accounts for all new operations, the analog of Lemma 5.1 of [10] holds (with the same proof): The dynamic tree algorithm runs in time $O((n_T m + n_{QA}) \log k)$, where n_{QA} is the total number of times a vertex is added to the queue of overflowing vertices.

There are $O(n_T n)$ passes over the queue. The argument is essentially the same as [10]: When an overflowing vertex v is chosen from the head of the queue (line 5 of `Min_VC_Split`) it is processed until either its excess becomes 0 or `Relabel`(v) is done, which either makes v dormant or increases $d(v)$. Define the potential function of [10], $\Phi = \max\{d(v) : v \in W \text{ is overflowing}\}$. If a pass over the queue increases Φ some distance $d(v)$ increases by at least the same amount. If a pass does not change Φ some distance $d(v)$ increases (since any vertex x with $d(x) = \Phi$ at the start of the pass either gets 0 excess or becomes dormant). Thus $O(n_T n)$ passes increase Φ or keep it the same.

It remains to show that $O(n_T n)$ passes decrease Φ. The number of passes that decrease Φ is at most the total increase in Φ. We have shown that all passes increase Φ by a total amount $O(n_T n)$. In addition Φ increases by a total amount $O(n_T n)$ when dormant sets are awakened, by the Transfer Bound. Hence the total increase in Φ is $O(n_T n)$, as desired.

The analog of Lemma 5.2 of [10] is that $n_{QA} = O(n_T(m + n^2/k))$. The proof of [10] applies without change. Now taking $k = n^2/m$ achieves time $O(n_T m \log(n^2/m))$ for `Min_VC_Split`.

This time bound suffices for the weighted connectivity algorithms of Section 2. It is desirable to replace n by n_T in the logarithmic factor for applications where n_T is appreciably smaller. (This includes the graphs generated by our reduction from the general weighted vertex cover split problem to the bipartite case, since the reduction can add $\Theta(m)$ new vertices.) To achieve this improved time bound we incorporate the bipush approach of [2] into our dynamic tree algorithm. We change our additional *send* operations into *bi-sends*. We do not change our additional *cut-children* operations (since performing *cut-children* for the root of a dynamic tree does not create any new leaves of nontrivial dynamic trees). All other changes are as specified in [2]. The analysis is the same as [2].

Theorem 3.3 *A minimum s-split of a vertex cover T can be found in time $O(n_T m \log(2+n_T^2/m))$ in a capacitated graph, for $n_T = |T| \leq n$. The space is $O(m)$.* □

References

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin: Network flows: Theory, Algorithms, and Applications; Prentice Hall, Englewood Cliffs, NJ, 1993

[2] R.K. Ahuja, J.B. Orlin, C. Stein, R.E. Tarjan: Improved algorithms for bipartite network flow. *SIAM J. Comput., 23,* 5, 1994, pp. 906–933.

[3] M. Becker, W. Degenhardt, J. Doenhardt, S. Hertel, G. Kaninke, W. Keber, K. Melhorn, S. Näher, H. Rohnert, T. Winter: A probabilistic algorithm for vertex connectivity of graphs. *Inf. Proc. Letters, 15,* 1982, pp. 135–136.

[4] J. Cheriyan, J.H. Reif: Directed s-t numberings, rubber bands, and testing digraph k-vertex connectivity. *Combinatorica, 14*, 4, 1994, pp. 435–451.

[5] D. Coppersmith, S. Winograd: On the asymptotic complexity of matrix multiplication. *SIAM J. Comput., 11,* 1982, pp. 472–492.

[6] S. Even: An algorithm for determining whether the connectivity of a graph is at least k. *SIAM J. Comput., 4,* 3, 1975, pp. 393–396.

[7] S. Even, R.E. Tarjan: Network flow and testing graph connectivity. *SIAM J. Comput., 4,* 4, 1975, pp. 507–518.

[8] Z. Galil: Finding the vertex connectivity of graphs. *SIAM J. Comput., 9,* 1, 1980, pp. 197–199.

[9] H.N. Gabow: Efficient splitting off algorithms for graphs. *Proc. 26th Annual ACM Symp. on Theory of Comp.,* 1994, pp. 696–705.

[10] A.V. Goldberg, R.E. Tarjan: A new approach to the maximum-flow problem. *J. ACM, 35,* 4, 1988, pp. 921–940.

[11] G. Gallo, M.D. Grigoriadis, R.E. Tarjan: A fast parametric maximum flow algorithm and applications. *SIAM J. Comput., 18,* 1, 1989, pp. 30–55.

[12] M.R. Henzinger: A static 2-approximation algorithm for vertex connectivity and incremental approximation algorithms for edge and vertex connectivity. *J. Algorithms, 24,*1,1997, pp. 194–220.

[13] J. Hao, J.B. Orlin: A faster algorithm for finding the minimum cut in a directed graph. *J. Algorithms, 17,* 3, 1994, pp. 424–446.

[14] V. King, S. Rao, R. Tarjan: A faster deterministic maximum flow algorithm. *J. Algorithms, 17,* 3, 1994, pp. 447–474.

[15] L. Lovász: Combinatorial Problems and Exercises, 2nd Ed., North-Holland, NY, 1993

[16] N. Linial, L. Lovász, A. Wigderson: Rubber bands, convex embeddings and graph conectivity. *Combinatorica, 8*, 1, 1988, pp. 91–102.

[17] H. Nagamochi, T. Ibaraki: A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica, 7,* 1992, pp. 583–596.