Using Expander Graphs to Find Vertex Connectivity

HAROLD N. GABOW

University of Colorado, Boulder, Colorado

Abstract. The (vertex) connectivity κ of a graph is the smallest number of vertices whose deletion separates the graph or makes it trivial. We present the fastest known algorithm for finding κ. For a digraph with n vertices, m edges and connectivity κ the time bound is $O((n + \min\{\kappa^{5/2}, \kappa n^{3/4}\})m)$. This improves the previous best bound of $O((n + \min\{\kappa^3, \kappa n\})m)$. For an undirected graph both of these bounds hold with m replaced by κn. Expander graphs are useful for solving the following subproblem that arises in connectivity computation: A known set R of vertices contains two large but unknown subsets that are separated by some unknown set S of κ vertices; we must find two vertices of R that are separated by S.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.2 [**Discrete Mathematics**]: Graph Theory

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: Expander graphs, graphs, vertex connectivity

1. *Introduction*

The *(vertex) connectivity* κ of a graph is the smallest number of vertices whose deletion separates the graph or leaves only one vertex. (Other basic terminology is defined at the end of this section.) This is a central concept of graph theory [Lovász 1993]. Computing the connectivity is posed as Research Problem 5.30 in Aho et al. [1974] where in fact a linear-time algorithm is conjectured. Yet relatively little progress has been made on computing connectivity.[1] If the conjectured linear-time algorithm exists, it involves techniques that are radically different from the known ones.

[1]This contrasts with a fairly large number of diverse and efficient algorithms for computing edge connectivity.

A preliminary version of this article appeared in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science,* IEEE Computer Society Press, Los Alamitos, CA, 2000, pp. 410–420.

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

Here we present the most efficient known algorithm for computing graph connectivity. We first summarize the results most relevant to ours. Other work on vertex connectivity, including randomized algorithms, is surveyed in Henzinger et al. [2000]. To *compute the connectivity* means to find κ and a corresponding separator of κ vertices. Throughout this article, unless noted otherwise, n and m denote the number of vertices and edges of the given graph, respectively.

Henzinger et al. [2000] give an algorithm to compute the connectivity of a digraph in time $O(\min\{\kappa^3 + n, \kappa n\}m)$. This algorithm applies two different high-level approaches to finding the connectivity, previously discovered by Even and Tarjan [1975] and Even [1975]. Henzinger et al. [2000] gains efficiency by using a preflow-push routine for computing rooted connectivity. For undirected graphs the above time bound improves using the maximal forest decomposition of Nagamochi and Ibaraki [1992]. This allows the graph to be pruned to $O(\kappa n)$ edges. So the time bound of Henzinger et al. [2000] for undirected graphs is the above bound with m replaced by κn.

Our new algorithm runs in time $O((n + \min\{\kappa^{5/2}, \kappa n^{3/4}\})m)$ for digraphs. For undirected graphs m can be replaced by κn in this bound. To compare our digraph bound with Henzinger et al. [2000] observe that both bounds are $O(nm)$ when $\kappa \leq n^{1/3}$. For larger values of κ, the new algorithm is faster: For $n^{1/3} \leq \kappa \leq \sqrt{n}$, Henzinger et al. [2000] is $O(\kappa^3 m)$; in this range, the new bound remains $O(nm)$ for $\kappa \leq n^{2/5}$ and becomes $O(\kappa^{5/2}m)$ for $\kappa \geq n^{2/5}$. For $\kappa \geq \sqrt{n}$, Henzinger et al. [2000] is $O(\kappa nm)$ and the new bound is $O(\kappa n^{3/4}m)$.

To *check k-connectedness* for a given k means to verify that $\kappa \geq k$ or else find κ and a corresponding separator of κ vertices. Let δ denote the minimum degree of the given graph. Henzinger [1997] uses maximal forest decompositions to check $\delta/2$-connectedness of an undirected graph. The time is $O(\min\{\kappa, \sqrt{n}\}n^2)$. Henzinger's algorithm is also based on the relation $\delta - \kappa \geq \kappa$ that holds trivially when $\kappa \leq \delta/2$. Our work (for both finding connectivity and checking k-connectedness) investigates general properties of this "gap" quantity $\delta - \kappa$. (Recall that we always have $\delta \geq \kappa$, since removing the neighbors of any vertex disconnects or trivializes any graph.) We show how to check $a\delta$-connectedness, where a is an arbitrary fixed constant <1, in time $O((\kappa + \sqrt{n})\sqrt{n}m)$ for digraphs and the same bound with m replaced by κn for undirected graphs. This is faster than our general connectivity finding algorithm. In other words, we can find the connectivity faster when κ/δ is bounded away from 1.

Our method is based on a nesting property of certain separation triples in undirected graphs, and its generalization to digraphs. This property eventually leads to a two step approach to computing vertex connectivity: The first step enlarges the gap $\delta - \kappa$ to some guaranteed size. (Initially the gap can be 0.) The second step is an algorithm to find the connectivity of a graph with a large gap.

Expander graphs play a crucial role in the second step. There are many diverse important applications of expander graphs (see, e.g., the lists in Alon [1986] and Wigderson and Zuckerman [1993], and the examples of Alon et al. [1994] and Spielman [1995]) but we are unaware of similar applications to graph algorithms.

Here's a simple example illustrating the connection between expanders and finding vertex connectivity: We wish to check that an undirected graph $G = (V, E)$ has connectivity $\geq n/30$. Somehow we've reduced this problem to checking that every set of exactly $n/2$ vertices has $\geq n/30$ neighbors. Previous algorithms don't take advantage of this extra knowledge. The best such algorithm uses time $O(n^2m)$

in this situation [Henzinger et al. 2000]. We solve the reduced problem in time $O(n^{3/2}m)$ using an expander as follows.

For simplicity, we'll use the Gabber–Galil expanders of degree 7. They have an expansion factor $\alpha = (2-\sqrt{3})/2 > 0.13$ [Motwani and Raghavan 1995]. Let X be such an expander on the vertex set V. The existence of this X requires that n be a perfect square. This doesn't present a problem, but for the purposes of this example assume n is a perfect square. So X is a graph of $\leq 7n$ edges where by definition, every set of exactly $n/2$ vertices has $\geq \alpha n/4 > n/30$ neighbors.[2]

Now suppose A is a set of $n/2$ vertices having $<n/30$ neighbors in G. Some edge (v, w) of X joins a vertex $v \in A$ to a vertex $w \notin A$ where w is a nonneighbor of A in G. Certainly v and w can be separated by $<n/30$ vertices in G (e.g., the neighbors of A in G are such a separating set). We can test if v and w are so separated by computing a maximum vw-flow in (a graph derived from) G.

So we can solve the reduced problem by computing such a vw-flow for every edge (v, w) of X, and checking that all flow values are $\geq n/30$. This solution computes just $O(n)$ max flows, achieving the desired time bound of $O(n^{3/2}m)$. (It's easy to construct the expander X in $O(n)$ time.)

In our two-step algorithm for computing connectivity (mentioned above), the first step of gap enlargement accomplishes essentially the reduction of the above example. The second step, handling graphs with a large gap, uses an expander construction similar to the example as a main tool (Lemmas 2.6–2.8). This construction is slightly more powerful than the simple approach of the example. It is based on the so-called Expander Mixing Lemma [Hoory et al. 2006]. The second step also depends on a nesting property (the Nesting Lemma 2.4) that governs the relationship between the complements of certain easily found separators and optimum separators.

As expected, there is a tradeoff in time between the two steps of our algorithm: In the first step, creating a larger gap takes more time. But in the second step, the larger the gap the faster the graph's connectivity can be found. This leads to a presentation of our algorithm in three parts: the gap enlargement step, the connectivity computation step, and the overall algorithm that chooses the size of the gap leading to the most efficient computation.

Although our asymptotic time bounds for undirected graphs are implied by the bounds for digraphs, we present a separate algorithm for undirected graphs. The reason is that undirected graphs have a stronger nesting property, which results in a simpler algorithm.

The article is organized as follows. Section 2 presents the connectivity algorithm for undirected graphs that already have large gaps. This includes the expander construction. Section 3 gives the undirected gap enlargement algorithm (i.e., the algorithm that sets the stage for Section 2). Sections 4–5 give the analogous results for digraphs: Section 4 first reviews digraph fundamentals, and then presents the digraph gap enlargement algorithm. Section 5 presents the more involved connectivity algorithm for digraphs that have large gaps. It also (in Section 5.4) presents our algorithm to check $a\delta$-connectedness. Section 6 combines the gap enlargement algorithm and the large gap connectivity algorithm, by choosing an appropriate

[2] The exact constants given here account for various technicalities—see the definition of (n, d, c)-expander, Section 2, and Motwani and Raghavan [1995].

gap size, to get our overall connectivity algorithm (for both directed and undirected graphs). Readers interested only in undirected graphs can skip Sections 4–5. Section 7 gives some brief conclusions. The rest of this section gives notation, definitions and some background.

A fraction written a/bc is an abbreviation for $a/(bc)$, e.g., $r^3/\rho\rho'$. The function $\log n$ denotes logarithm base two. We assume $\log 1 = 1$ (some of our time bounds have a logarithm in the denominator). We usually denote singleton sets $\{v\}$ by v, as in $S - v$. Consider a graph $G = (V, E)$. For the rest of this section, assume G is undirected; the minor changes needed when G is directed are discussed in Section 4. If H is not the given graph, we usually refer to its vertex and edge set as $V(H)$ and $E(H)$ respectively. More generally if a function refers to a graph, we include the graph as an extra argument if it is not clear, for example, $n(H)$. δ denotes the minimum degree of a vertex of G. A *neighbor* of a set of vertices X is a vertex $y \notin X$ on an edge (x, y) for some $x \in X$. A *nonneighbor* of X is a vertex $y \notin X$ that is not a neighbor.

A *separation triple* (S, X, Y) is an ordered triplet of sets forming a partition of V, with X and Y nonempty and no edge going from X to Y. S is the *separator* and X and Y are the *shores* of the triple.

A separator does not uniquely determine the shores of a corresponding separation triple. For instance, a star with center c and leaves L has separation triples $(\{c\}, X, L - X)$ for every set $X \neq \emptyset, L$. On the other hand, if X is a shore of a separation triple then the set of neighbors of X is the smallest possible corresponding separator.

The *vertex connectivity* κ is the smallest cardinality of a separator, or $n - 1$ if the graph is complete. A *κ-separation triple* has $|S| = \kappa$. In that case, S is a *κ-separator* and X and Y are *κ-shores*. We usually star the sets of a κ-separation triple as in (S^*, X^*, Y^*) to call attention to their optimality. As mentioned, any graph has $\delta \geq \kappa$.

For $x, y \in V$,

$$\kappa(x, y) = \min\{|S|,\ n - 1 : (S, X, Y) \text{ is a separation triple with } x \in X,\ y \in Y\}.$$

Note that $\kappa(x, y) = n-1$ exactly when $x = y$ or $(x, y) \in E$. The *rooted connectivity* at x is

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

When the graph G is not obvious we write $\kappa(x, y; G)$ or $\kappa(x; G)$ for these two notions, respectively.

We also use a notion introduced by Even [1975]: For a set of vertices S,

$$\kappa_W(S) = \min\{|S|, n-1, |C| : (C, A, B) \text{ is a separation triple of } G \text{ with } S \subseteq A \cup C\}.$$

(The "W" stands for weak separation.) To compute $\kappa_W(S)$ first form the graph $G_W(S)$ by starting with G and adding a new vertex s with edges (s, v), $v \in S$. Then $\kappa_W(S) = \min\{\kappa(s; G_W(S)),\ n - 1\}$.

Any quantity $\kappa(x, y)$ can be computed using one max flow computation on a network with unit vertex capacities [Ahuja et al. 1993]. The time is $O(\sqrt{n}m)$ (using the Hopcroft–Karp algorithm). Alternatively for any value k we can find $\kappa(x, y)$ or verify that $\kappa(x, y) \geq k$ in time $O(km)$ (using the Ford–Fulkerson algorithm).

Any quantity $\kappa(x)$ can be computed in $O(nm)$ time [Henzinger et al. 2000]. So the same bound holds for quantities $\kappa_W(S)$.

Finally, we review the maximal forest decomposition of Nagamochi and Ibaraki [1992]. Consider an undirected graph $G = (V, E)$. The algorithm of Nagamochi and Ibaraki [1992] partitions E, in $O(m)$ time, into a sequence of forests F_k, $k = 1, \ldots, n$. For $k = 0, \ldots, n$ define the subgraph FG_k by

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

Clearly, FG_k has $O(kn)$ edges. The construction actually makes each F_k ($k \geq 1$) a maximal forest of the graph $G - FG_{k-1}$, hence the name maximal forest decomposition. For every $k \geq 1$, G is k-connected if and only if FG_k is k-connected. In fact, Nagamochi and Ibaraki [1992] proves this stronger property: Any set of $<k$ vertices is a vertex separator of G if and only if it is a vertex separator of FG_k (see the proof of Theorem 3.1 in Nagamochi and Ibaraki [1992]).

2. *Gap Algorithm*

This section presents our main algorithm to check k-connectedness of an undirected graph. When we actually compute κ (Section 6) this Gap Algorithm gets called with an appropriate value of k that does the trick. The efficiency of the Gap Algorithm depends on having a large "gap" $\delta - k$. Section 2.1 establishes the graph properties that we need, and Section 2.2 presents the Gap Algorithm and its analysis.

2.1. FOUNDATIONS: RICH SETS AND EXPANDERS. The algorithm is organized around a nesting property (Lemma 2.4) that sets up the use of expanders.

We begin with a simple bound on the size of a shore of a κ-separation triple.

PROPOSITION 2.1. *Any κ-shore has $\geq \delta + 1 - \kappa$ vertices.*

PROOF. Let (C, A, B) be any κ-separation triple. We have $|A| + |C| > \delta$ because a vertex of A has all its neighbors in $A \cup C$. Hence $|A| \geq \delta + 1 - \kappa$. $\square$

A set $R \subseteq V$ is *ρ-rich* if it contains at least ρ vertices from each shore of some κ-separation triple. Hence $|R| \geq 2\rho$. Such an R is *ρ-superrich* if it contains at least ρ vertices from each shore of every κ-separation triple. As an example note how 1-rich sets allow us to compute the connectivity: A set of two vertices $\{x, y\}$ that is 1-rich has $\kappa = \kappa(x, y)$. An arbitrary set R that is 1-rich allows us to compute κ using $O(|R|^2)$ max flow computations. However, sets that are richer than 1-rich will be even more useful to us.

As a minor point, note that the complete graph has no rich or superrich sets, since there are no κ-separation triples.

Part (i) of the following lemma is our main tool for finding rich sets. Part (ii) is not used in this article and is included for completeness; however, its directed analog is used (Lemma 4.1).

LEMMA 2.2

(*i*) *The neighbors of a vertex v with $\kappa(v) > \kappa$ form a ρ-superrich set for $\rho = \min\{\kappa(v), \delta\} - \kappa + 1$.*

(*ii*) *Any set $S \subseteq V$ with $\kappa_W(S) > \kappa$ is ρ-superrich for $\rho = \min\{\kappa_W(S), \delta + 1\} - \kappa$.*

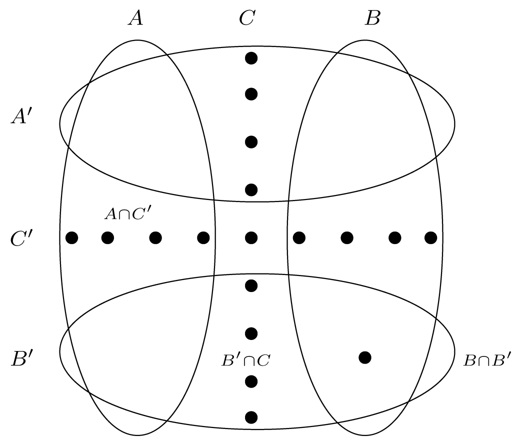

FIG. 1. Separation triples (C, A, B) and (C', A', B') in Lemma 2.3. The vertex set V is partitioned into the three sets A, B, C. It is also partitioned into A', B', C'.

PROOF. Note that both parts of the lemma are vacuous if the graph is complete, so assume the opposite. Let (C^*, A^*, B^*) be any κ-separation triple. Both parts (i) and (ii) of the lemma define ρ so that $\rho \leq \delta - \kappa + 1$. Hence, Proposition 2.1 shows that both parts (i) and (ii) have

$$|A^*| \geq \rho. \tag{1}$$

(i) The hypothesis $\kappa(v) > \kappa$ forces C^* to contain v. Let S be the set of neighbors of v and assume that contrary to the lemma, $|A^* \cap S| < \rho$. (1) gives $A^* - S \neq \emptyset$. This ensures that

$$(C^* \cup (A^* \cap S) - v,\ A^* - S,\ B^* \cup v)$$

is a separation triple. Now the definitions of $\kappa(v)$ and ρ imply a contradiction:

$$\kappa(v) \leq |C^* \cup (A^* \cap S) - v| = |C^*| + |A^* \cap S| - 1 < \kappa + \rho - 1 \leq \kappa(v).$$

(ii) The argument is similar. Assume that contrary to the lemma, $|A^* \cap S| < \rho$. Again (1) gives $A^* - S \neq \emptyset$. This ensures that

$$(C^* \cup (A^* \cap S),\ A^* - S,\ B^*)$$

is a separation triple. Now the definitions of $\kappa_W(S)$ and ρ imply a contradiction:

$$\kappa_W(S) \leq |C^* \cup (A^* \cap S)| = |C^*| + |A^* \cap S)| < \kappa + \rho \leq \kappa_W(S). \quad \square$$

Consider two arbitrary separation triples (C, A, B) and (C', A', B') as illustrated in Figure 1. (Some features of the figure are oriented towards Lemma 2.3.) The figure naturally suggests that the neighbors of a "quadrant" (like $A \cap A'$) are contained in parts of the two separators, for example,

the neighbors of $A \cap A'$ are contained in $(A' \cap C) \cup (C' - B)$.

To prove this observe that no edge joins A to B, and no edge joins A' to B'. Hence a neighbor x of $A \cap A'$ does not belong to $(A \cap A') \cup B \cup B'$. So $x \in A'$ implies $x \in A' \cap C$ and $x \notin A'$ implies $x \in C' - B$.

A shore B of a separation triple (C, A, B) is *extreme* if every nonempty subset of B has at least $|C|$ neighbors.[3] Here are three examples: The shore B of separation

[3] A similar notion of "extreme set" is important for edge connectivity [Frank 1994, p. 51].

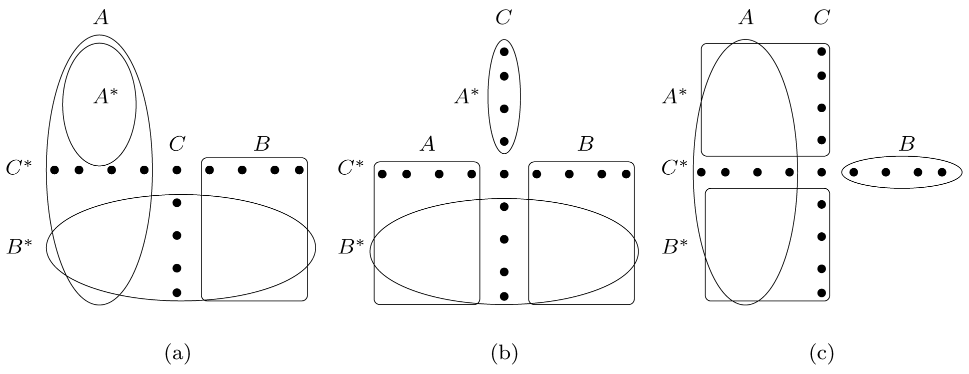

FIG. 2. The possible relations given by the Nesting Lemma 2.4. (C, A, B) is a separation triple with B extreme. (C^*, A^*, B^*) is a κ-separation triple implied by the lemma. The lemma's part (i) is illustrated in (a)–(b), and part (ii) is illustrated in (c). (c) shows the possibility of B contained in the κ-separator C^*.

triple (C, A, B) is extreme if (C, A, B) is either

a κ-separation triple, or
a separation triple corresponding to $\kappa(x)$, more precisely $x \in A$ and $|C| = \kappa(x)$,
or a separation triple corresponding to $\kappa_W(S)$, more precisely $S \subseteq A \cup C$ and $|C| = \kappa_W(S)$.

So the separation triples we're concerned with have an extreme shore. The next lemma on extreme shores is illustrated in Figure 1.

LEMMA 2.3. *Let (C, A, B) and (C', A', B') be separation triples with B' an extreme shore and $B \cap B' \neq \emptyset$.*

(i) $|B' \cap C| \geq |A \cap C'|$.
(ii) *$A \cap A'$ has at most $|C|$ neighbors.*

Remark. In (ii), it is possible that $A \cap A'$ is empty.

PROOF

(i) Since B' is extreme, $B \cap B'$ has at least $|C'|$ neighbors. Applying our characterization of the neighbors of quadrant $B \cap B'$ shows this means $|B' \cap C| + |C' - A| \geq |C'|$. This implies (i).
(ii) We have shown the neighbors of quadrant $A \cap A'$ are contained in $(A \cap C') \cup (C - B')$. By (i) the size of this set is $|A \cap C'| + |C - B'| \leq |B' \cap C| + |C - B'| = |C|$. □

We apply the lemma to prove the following nesting property. It allows us to either locate a κ-separator or find a rich set. It is illustrated in Figure 2. (In Figure 2(a), it is easy to see that $B \cap C^* = \emptyset$ always holds, but we do not use this property.)

LEMMA 2.4 (NESTING LEMMA). *Let (C, A, B) be a separation triple with B an extreme shore. Either*

(i) *A or C contains a κ-shore, or*
(ii) *A or B is contained in a κ-separator, and C is* $\min\{|A|, |B|\}$*-rich.*

PROOF. Let (C^*, A^*, B^*) be a κ-separation triple. We can assume

$$A^* \nsubseteq C \quad \text{and} \quad B^* \nsubseteq C \tag{2}$$

since otherwise the second alternative of (i) holds.

The argument divides into two cases depending on whether or not C^* satisfies the first part of (ii). We first assume it does not.

Case (1). $A \nsubseteq C^*$ and $B \nsubseteq C^*$.

After possibly interchanging sets A^* and B^*. we can assume

$$A \cap A^* \neq \emptyset \quad \text{and} \quad B \cap B^* \neq \emptyset.$$

To see this choose A^* so that $A \cap A^* \neq \emptyset$ by the first assumption of Case (1). If this makes $B \cap B^* \neq \emptyset$, we're done so assume the opposite. Then $B \cap A^* \neq \emptyset$ by the second assumption of Case (1) and $A \cap B^* \neq \emptyset$ by the second part of (2). Interchange A^* and B^* and again we're done.

Since B is extreme and $B \cap B^* \neq \emptyset$, Lemma 2.3(ii) shows that $A \cap A^*$ has at most $|C^*| = \kappa$ neighbors. Since $A \cap A^*$ is nonempty and it has nonneighbors (e.g., $B \cap B^*$), it is thus a κ-shore. Now the first alternative of (i) holds.

Case (2). $A \subseteq C^*$ or $B \subseteq C^*$.

This case assumes the first part of (ii) so we need only show the second part. The argument will not use the extremeness of B, that is, it is symmetric in A and B. So, without loss of generality, assume

$$B \subseteq C^*.$$

This implies neither of the sets A^*, B^* intersects B, so (2) implies $A^* \cap A,\ B^* \cap A \neq \emptyset$. Since A^* is extreme and $A \cap A^* \neq \emptyset$, Lemma 2.3(i) shows that $|A^* \cap C| \geq |B \cap C^*| = |B|$. Similarly, since B^* is extreme and $A \cap B^* \neq \emptyset$, Lemma 2.3(i) shows that $|B^* \cap C| \geq |B \cap C^*| = |B|$. These two inequalities plus the symmetry of A and B gives the second part of (ii). $\square$

Expanders allow us to find κ once we have a sufficiently rich set. We now establish the connection. To maintain consistency with the literature in the rest of Section 2.1, n denotes the number of vertices in the various graphs we consider.

Recall that an *(n, d, c)-expander* is a d-regular bipartite multigraph (V, W, F) where $|V| = |W| = n/2$ such that any subset $S \subseteq V$ has at least $(1 + c(1 - 2|S|/n))|S|$ neighbors [Motwani and Raghavan 1995, p. 145]. Our techniques can be applied using linear-sized (n, d, c) expanders, such as the graphs of Gabber and Galil [Motwani and Raghavan 1995]. This leads to a vertex connectivity algorithm with the same time bounds as achieved in this article for graphs with $\kappa \leq n^{3/4}$. But to achieve the full range of our results (which allow $\kappa \leq n$) we rely on graphs with a stronger expansion property.

Consider a connected undirected multigraph G that is regular of degree d. Let $A(G)$ be the adjacency matrix of G. Recall these facts about the eigenvalues of $A(G)$ [Motwani and Raghavan 1995]: Every eigenvalue has magnitude at most d. The largest eigenvalue is in fact d, and its multiplicity is one. G is bipartite if and only if $-d$ is an eigenvalue, if and only if $-d$ is an eigenvalue of multiplicity one.

We use the family of Ramanujan graphs constructed by Lubotzky et al. [1988] and Margulis [1988] (see also Davidoff et al. [2003]). They showed the following:

LEMMA 2.5 (LUBOTZKY, PHILLIPS, SARNAK; MARGULIS). *There is a constant $C_0 > 1$ such that for any integers n_0 and d_0, there are integers $\nu \in [n_0, C_0 n_0]$ and $\delta \in [d_0, C_0 d_0]$ and a connected regular graph $X_{\nu,\delta}$ having ν vertices and degree δ, each of whose eigenvalues other than $\pm\delta$ has magnitude $\le 2\sqrt{\delta - 1}$.*

Furthermore, the graphs $X_{\nu,\delta}$ can be constructed in time and space linear in their size, $O(\nu\delta)$.

$X_{\nu,\delta}$ may be bipartite or not, but the two cases behave similarly. We discuss the nonbipartite case first. The following proposition, which encapsulates our use of expanders, is an immediate consequence of a result of Alon et al. [1992, p. 122, Corollary 2.5]; see also Hoory et al. [2006].

LEMMA 2.6. *Suppose $X_{\nu,\delta}$ is nonbipartite. Then any two sets $A, B \subseteq V(X_{\nu,\delta})$ are joined by an edge of $X_{\nu,\delta}$ if*

$$|A|\,|B|\,\delta \ge 4\nu^2.$$

PROOF. First consider an arbitrary connected multigraph G. For any two (not necessarily disjoint) sets of vertices A, B, let $E(A, B)$ denote the number of ordered pairs (a, b) with $a \in A,\ b \in B$ and (a, b) an edge of G. Let $\lambda(G)$ be the largest magnitude of any eigenvalue of $A(G)$ aside from the maximum degree of G. The above-cited result of Alon et al. [1992] states that for any regular degree d graph G on n vertices and any two sets of vertices A, B,

$$|\, E(A, B) - |A||B|d/n \,| \le \lambda(G)\sqrt{|A||B|}. \tag{3}$$

Now consider $X_{\nu,\delta}$. If it is nonbipartite, $-\delta$ is not an eigenvalue. Hence, Lemma 2.5 shows $\lambda(X_{\nu,\delta})^2 < 4\delta$. So the present lemma's hypothesis implies $|A||B|\,\delta^2 \ge 4\delta\nu^2 > (\lambda(X_{\nu,\delta})\nu)^2$. Taking square roots and manipulating gives $|A|\,|B|\,\delta > \lambda(X_{\nu,\delta})\,\nu\sqrt{|A||B|}$. Now (3) applied to $X_{\nu,\delta}$ implies $E(A, B) > 0$. □

Next, we modify $X_{\nu,\delta}$ so the lemma holds in the bipartite case as well. Suppose $X_{\nu,\delta}$ is bipartite, say with bipartition V_0, V_1. The construction of Lubotzky et al. [1988] and Margulis [1988] has $|V_0| = |V_1|$ (since $X_{\nu,\delta}$ is defined as a Cayley graph). Construct an arbitrary bijection between V_0 and V_1, and contract each pair of corresponding vertices $v_0 \in V_0$, $v_1 \in V_1$. Call the new graph $X'_{\nu,\delta}$. It has $\nu/2$ vertices and maximum degree 2δ. It satisfies the following analog of Lemma 2.6:

COROLLARY 2.7. *Suppose $X_{\nu,\delta}$ is bipartite. Then, any two disjoint sets $A, B \subseteq V(X'_{\nu,\delta})$ are joined by an edge of $X'_{\nu,\delta}$ if*

$$|A|\,|B|\,\delta \ge \nu^2.$$

PROOF. The proof is essentially identical to the preceding lemma. Consider an arbitrary connected bipartite multigraph G with bipartition V_0, V_1. Take sets $A \subseteq V_0$ and $B \subseteq V_1$. Define $E(A, B)$ as above. Let $\lambda(G)$ be the largest magnitude of any eigenvalue of $A(G)$ aside from $\pm d$. The analog of (3) is that for any regular degree d bipartite graph G on n vertices,

$$|\, E(A, B) - 2|A||B|d/n \,| \le \lambda(G)\sqrt{|A||B|}.$$

(This inequality is also given in Motwani and Raghavan [1995, p. 160, Ex. 6.27]; see also the proof of (3) in Hoory et al. [2006].)

Now consider $X_{\nu,\delta}$ of the Corollary, with bipartition V_0, V_1. For the disjoint sets A, B of the Corollary and for $i = 0, 1$, let A_i (B_i) be the vertices V_i corresponding to vertices of A (B), respectively. Lemma 2.5 shows $\lambda(X_{\nu,\delta})^2 < 4\delta$. So the present lemma's hypothesis implies $4|A_0||B_1|\,\delta^2 \geq 4\delta\nu^2 > (\lambda(X_{\nu,\delta})\nu)^2$. Taking square roots and manipulating gives $2|A_0||B_1|\,\delta > \lambda(X_{\nu,\delta})\nu\sqrt{|A_0||B_1|}$. Now the displayed inequality applied to $X_{\nu,\delta}$ implies $E(A_0, B_1) > 0$. An edge joining A_0 and B_1 does not get contracted when we form $X'_{\nu,\delta}$ (by the disjointness of A and B). Hence A and B are joined by an edge of $X'_{\nu,\delta}$. $\square$

We return to the given graph $G = (V, E)$ of the connectivity computation. For any set R of r vertices of V and any positive integer d, we will define a graph $\chi(R, d) = (R, F)$. The vertex set is R, so we need only specify the edge set F, which we do as follows.

Take $n_0 = 2r$ and $d_0 = d$ and construct the expander graph $X_{\nu,\delta}$ of Lemma 2.5. If $X_{\nu,\delta}$ is nonbipartite identify each vertex of R with a unique vertex of $X_{\nu,\delta}$, and let F consist of the edges induced by $X_{\nu,\delta}$ on R. If $X_{\nu,\delta}$ is bipartite identify each vertex of R with a unique vertex of $X'_{\nu,\delta}$, and let F consist of the edges induced by $X'_{\nu,\delta}$ on R. (Note the identification is possible since $X'_{\nu,\delta}$ has $\nu/2 \geq 2r/2 = r$ vertices.)

LEMMA 2.8 (EXPANDER LEMMA). *Take any $k \geq \kappa$ and let R be a ρ-rich set of r vertices. Set $\rho' = \max\{\rho,\ (r-k)/2\}$ and $d = (4C_0r)^2/(\rho\rho')$. Then*

$$\kappa = \min\{\kappa(x, y) : (x, y) \text{ an edge of } \chi(R, d)\}.$$

Remark. The parameter k occurs because we will use the lemma to check k-connectedness. The reason for ρ' will be seen in the proof of Lemma 2.10. Note also that d is an integer ≥ 16, since $r \geq \rho, \rho'$. In particular, $d = O(r^2/\rho\rho')$.

PROOF. The ρ-richness of R shows there is a κ-separation triple (C^*, A^*, B^*) such that, writing $\alpha = |A^* \cap R|$ and $\beta = |B^* \cap R|$, we have $\alpha \geq \beta \geq \rho$. We also have $\alpha \geq \rho'$ since $|R \cap C^*| \leq \kappa \leq k$ implies $\alpha \geq (r-k)/2$. Since the definition of $\chi(R, d)$ has $\delta \geq d$ and $\nu \leq 2C_0r$, we get $\alpha\beta\delta \geq \rho'\rho d \geq 4(C_02r)^2 \geq 4\nu^2$. Thus, Lemma 2.6 or Corollary 2.7 implies $\chi(R, d)$ has an edge (x, y) joining vertices of A^* and B^*. (For Corollary 2.7, observe that A^* and B^* are disjoint sets.) So $\kappa(x, y) = \kappa$. $\square$

Observe that $\chi(R, d)$ has r vertices, and maximum degree at most δ (2δ) in the nonbipartite (bipartite) case, respectively. So the maximum degree of $\chi(R, d)$ is always $O(d) = O(r^2/\rho\rho')$, and the number of edges is

$$m(\chi(R, d)) = O(r^3/\rho\rho'). \tag{4}$$

(Recall our convention for writing denominators of fractions, stated in Section 1.) Given R, ρ and k, the time to compute κ using the lemma is dominated by the time for $m(\chi(R, d))$ max flow computations of values $\kappa(x, y)$ on the given graph G.

We close this section by peeking ahead to see how the overall connectivity algorithm uses this expander construction. Table I shows the size of $\chi(R, d)$ in all its different uses. These values do not appear explicitly in our derivation; we show them here to give the reader a better feel for the algorithm.

Row 1 of Table I is for our algorithm to check $a\delta$-connectedness (a is a constant, $0 < a < 1$; ignoring constant factors, the parameter k is δ if $\delta < \sqrt{n}$, and otherwise

TABLE I. SIZE OF THE EXPANDER GRAPH $\chi(R, d)$, AS USED IN THE OVERALL CONNECTIVITY ALGORITHM

Row	Range	Number Vertices	Degree	Time for $\kappa(x, y)$
1	$k \le a\delta$	k	$O(1)$	$O(\sqrt{n}m)$
2	$\delta \le \sqrt{n}$	δ	$\sqrt{\delta}$	$O(\delta m)$
3	$\sqrt{n} < \delta \le n^{2/3}$	δ	$n^{1/4}$	$O(\sqrt{n}m)$
4	$n^{2/3} < \delta \le n/3$	δ	$n^{1/4} \log_{n/\delta} n$	$O(\sqrt{n}m / \log_{4n/\delta} n)$
5	$n/3 < \delta \le n - n^{7/8}$	δ	$\dfrac{n^{5/4} \log n}{n - \delta}$	$O\left(n + \dfrac{(n-\delta)^{5/2}}{\log(n-\delta)}\right)$

Constant factors are ignored. The third column gives $r = n(\chi(R, d))$ and the fourth column gives the degree d of the expander. Row 1 is for the algorithm to check $a\delta$-connectedness (Section 5.4). Rows 2–5 are for the Gap Algorithm as it is used in the Main Algorithm (Section 6); each row corresponds to a different range of δ (column 1) as well as a different bound for the network flow computation of $\kappa(x, y)$ (column 5).

k takes on values in the range $\sqrt{n} \le k \le \delta$.) This algorithm is also used as the first step in our general connectivity algorithm. The remaining rows all correspond to the Gap Algorithm, as it is invoked by the general connectivity algorithm (Section 6). The degree of the expander that we use is affected by the time for a network flow computation of the quantity $\kappa(x, y)$, as shown by the last column of Table I. This is not surprising considering Lemma 2.8. Row 2 corresponds to the time bound of $O(\delta m)$ to compute a flow having the largest possible value $\le \delta$. Row 3 is the standard time bound for computing a maximum flow in a unit capacity graph. (Both these bounds are recalled in Section 1.) Rows 4–5 correspond to time bounds for flow in very dense graphs (see Proposition 2.9). The time bound of our overall connectivity algorithm, $O((n + \min\{\kappa^{5/2}, \kappa n^{3/4}\})m)$, is essentially the product of columns 3–5 in rows 2–5 (ignore the nm term in the time bound, change δ to κ, and in the last row replace $n - \delta$ by n).

Sometimes we need to tailor our algorithms to take advantage of these different flow bounds (Flow Version of the Gap Enlarger, Section 3; Flow Version of the Directed Gap Algorithm, Section 5.3). Notice also that the expander $\chi(R, d)$ is always smaller than the given graph G, both in terms of number of vertices and number of edges. The values of Table I can be verified from the proofs of Theorem 5.5, Lemma 6.1, Lemma 6.2 Cases (1), (2), and (3), respectively, although this is not necessary for the derivation of our results.

2.2. ALGORITHM.

This section combines the tools of the last section to get the Gap Algorithm.

All the algorithms in this article calculate a number of separation triples. They maintain (C_0, A_0, B_0) as a triple having the smallest separator C_0 seen so far. We say the algorithm *has found* κ when $|C_0| = \kappa$. It is not necessary for the algorithm to know that the equation holds at this point. So $|C_0| > \kappa$ as long as the algorithm has not found κ.

The Gap Algorithm checks k-connectedness, that is, given k it finds a κ-separator if $\kappa < k$ or it determines that $\kappa \ge k$. Define the "gap" γ, and the "half-gap" τ, by

$$\gamma = \delta - k, \quad \tau = \gamma/2.$$

The Gap Algorithm assumes $\gamma \ge 4$. (This assumption can be weakened but it suffices for our purposes.) No other assumptions are made on the input graph or k.

As we present the algorithm, we also verify its correctness. The explanations of each step that verify correctness collectively show that if $\kappa < k$ a κ-separator will be found by the algorithm. (Thus, if $\kappa < k$, the algorithm knows it has found κ when it halts.)

In brief, the Gap Algorithm works as follows. It proceeds on the assumption that $\kappa < k$. Steps (1)–(3) either find κ or find a τ-rich set R. The latter is done in various ways (in a trivial way in Step (1), or by applying Lemma 2.2(i) or the Nesting Lemma 2.4 in Step (3)). The τ-rich set R is processed in Step (4) (in all cases). Assuming $\kappa < k$ and κ still has not been found, Step (4) finds κ using the Expander Lemma 2.8. The larger the gap γ, the more efficient our algorithm becomes.

Gap Algorithm

Step (1). If $n = \delta + 1$, then the graph is k-connected so return. If $n < 2\delta$, then set R to V and go to Step (4).

Explanation. If $n = \delta + 1$, the graph is complete and certainly $\kappa = \delta = k + \gamma > k$. After this test, we can assume the graph is not complete, so separators exist. Now suppose $n < 2\delta$. Step (4) requires that R be τ-rich. We can show that V is τ-rich even without the inequality $n < 2\delta$: Assuming $\kappa < k$, Proposition 2.1 shows that any κ-shore has $\geq \delta + 1 - \kappa > \delta - k = \gamma > \tau$ vertices.

Step (2). Let a be a vertex of degree δ. Find $\kappa(a)$ and a corresponding separation triple (C, A, B) with $a \in A$. Initialize (C_0, A_0, B_0) to this triple.

Step (3). /* If $\kappa < k$ has not been found, this step either finds κ or makes R a τ-rich set of $\leq \delta$ vertices. */

Step (3.1). If $\kappa(a) \geq k + \tau$, then let R be the set of neighbors of a and go to Step (4).

Explanation. Lemma 2.2(i), with $\kappa < \kappa(a) \leq \delta$ and $\kappa(a) - \kappa + 1 > \tau$, implies that R is τ-rich. Also the choice of a ensures $|R| = \delta$.

Step (3.2). Consider the opposite case $\kappa(a) < k + \tau$. We will apply the Nesting Lemma 2.4 to the separation triple (C, A, B) corresponding to $\kappa(a)$. Let D be any set of k vertices in $A \cup B$. Compute $\kappa_W(C)$ and $\kappa_W(D)$ and corresponding separation triples and update (C_0, A_0, B_0). Set R to C and go to Step (4).

Explanation. The Nesting Lemma 2.4 is applicable since B is extreme. If part (i) of the Nesting Lemma holds, then Step (3.2) finds κ: If A contains a κ-shore, then $\kappa = \kappa_W(C)$ (since we can assume $|C| > \kappa$). If C contains a κ-shore, then $\kappa = \kappa_W(D)$. (Note that D exists since $n \geq 2\delta > 2k + \tau > k + |C|$.) The remaining possibility is Lemma 2.4(ii). Since every vertex has degree at least $\delta = k + 2\tau$ and $|C| = \kappa(a) < k + \tau$, we have $|A|, |B| \geq \delta + 1 - |C| > k + 2\tau - |C| > \tau$. So C is τ-rich and $|C| \leq \delta$ as desired.

Step (4) /* If $\kappa < k$ has not been found then R is now τ-rich and either $R = V$ with $n < 2\delta$ (from Step 1) or $|R| \leq \delta \leq n/2$ (from Step (3)). */

If $|R| \leq \delta \leq n/2$, enlarge R to exactly 2δ vertices by adding (arbitrary) vertices of G to R. Apply the Expander Lemma 2.8 with $\rho = \tau$, constructing the graph $\chi(R, d)$ and computing the values $\kappa(x, y)$ specified in the lemma. Update (C_0, A_0, B_0) for these values.

Explanation. If Step (4) starts with $\kappa < k$ and κ still not found, the Expander Lemma guarantees that Step (4) computes κ.

Having shown the algorithm is correct, we turn to analyzing its efficiency. Step (4) requires efficient computation of the connectivity quantities $\kappa(x, y)$. As reviewed in Section 1 one value $\kappa(x, y)$ can be computed in time $O(\sqrt{n}m)$. This bound can be improved on very dense graphs, as indicated in the next Proposition 2.9. For reasons of continuity we postpone the proof of the proposition: We first derive the time bounds for the Gap Algorithm (Lemma 2.10) invoking Proposition 2.9. Then we close the section by proving the proposition.

PROPOSITION 2.9. *In a directed or undirected graph G, any quantity $\kappa(x, y)$ can be computed in the following time bounds:*

(i) $O(\sqrt{n}m/\log_{4n/\delta} n)$.

(ii) $O(n + (n-\delta)^{5/2}/\log(n-\delta))$*, assuming G is given as an adjacency matrix.*

Remark. As shown in the proof of the proposition below, bound (ii) is implied by (i) when $\delta \leq n/2$. So (ii) is of interest only when $\delta > n/2$. In this case, an adjacency matrix for G still uses only linear ($O(m)$) space. For values of δ very close to n, the bound of (ii) is sublinear ($o(m)$).

To bound the time for the Gap Algorithm, we allow $O(m)$ time for bookkeeping. Beyond that, Steps (2)–(3) perform $O(1)$ computations of quantities $\kappa(x)$ and $\kappa_W(S)$. As mentioned in Section 1, this uses time $O(nm)$. So the total time is $O(nm)$ plus the time for Step (4).

Step (4) constructs $\chi(R, d)$ and performs $m(\chi(R, d))$ max flow computations of quantities $\kappa(x, y)$. (4) with $\rho = \tau = \Theta(\gamma)$ shows that the time amounts to

$$O(r^3/\gamma(r-k))$$

computations of $\kappa(x, y)$.

The next lemma presents four time bounds for the Gap Algorithm. (i) is the main bound. (ii) and (iii) are used when κ and δ are very close to n (Lemma 6.2). These two bounds assume the algorithm computes max flows using Proposition 2.9. (Some further simplification of the bounds of (i)–(iii) is possible, but we refrain from doing so, in order to achieve consistency with the directed version Lemma 5.2, and for a simpler overall development.) (iv) is used to derive our efficient k-connectedness algorithm (Theorem 5.5). In all four cases, the proof simply amounts to multiplying the number of flow computations displayed above by the time for one max flow computation.

LEMMA 2.10. *If $\gamma \geq 4$, then the Gap Algorithm either verifies that $\kappa \geq k$ or finds a κ-separator.*

(i) *If $n \geq 2\delta$, the time is $O((n + \frac{\delta^2}{\gamma}\min\{\delta, \sqrt{n}\})m)$.*

(ii) *Alternatively, if $n \geq 2\delta$, the time is $O((n + \frac{\delta^2\sqrt{n}}{\gamma \log_{4n/\delta} n})m)$.*

(iii) *In general, the time is $O(\frac{n^{9/2}}{\gamma\log(n-\delta)})$.*

(iv) *If $\gamma \geq \epsilon\delta$ for some fixed $\epsilon > 0$, the time is $O((\sqrt{n} + \delta)\sqrt{n}m)$.*

PROOF

(i)–(ii) We need only account for Step (4), since the remaining time $O(nm)$ is clearly within the desired bound.

The hypothesis $n \geq 2\delta$ shows Step (4) applies the Expander Lemma 2.8 with $r = 2\delta$. Since $2\delta \geq \delta + k$, the number of flow computations (as displayed above) is $O(\delta^3/(\gamma(2\delta - k))) = O(\delta^2/\gamma)$.

For (i), observe that flow values $\geq k$ are irrelevant to computing $\kappa < k$. In other words, it suffices that Step (4) computes $\max\{k, \kappa(x, y)\}$. As noted in Section 1 the time for such a max flow computation is $O(\min\{k, \sqrt{n}\}\, m)$. This time bound implies (i).

For (ii), Step (4) computes max flows using Proposition 2.9(i).

(*iii*) The time $O(nm) = O(n^3)$ is dominated by the desired time bound since we can obviously assume $n^{3/2} \geq n \log n \geq \gamma \log(n - \delta)$. Hence, we need only analyze the time for Step (4).

When $n < 2\delta$, we have $R = V$, $r = n$ and the number of flow computations (as displayed above) is $O(n^3/(\gamma(n - k))) = O(n^3/(\gamma(n - \delta)))$. This bound also holds when $n \geq 2\delta$ since for this case the proof of (i)–(ii) shows we compute $O(\delta^2/\gamma)$ flows. It suffices to compute each flow in time $O(n + (n - \delta)^{5/2}/\log(n - \delta))$. So the desired time bound follows using Proposition 2.9(ii) (we can certainly convert G to an adjacency matrix representation within the desired time bound).

(*iv*) It is easy to see that we need only show Step (4) does $O(\delta)$ max flow computations. (4) shows the number of flow computations is $O(r^3/\rho\rho')$. Step (4) always has $r \leq 2\delta$. Also $\rho' \geq \rho = \tau \geq \epsilon\delta/2$. So the number of flow computations is $O(\delta^3/(\epsilon\delta)^2) = O(\delta)$. □

It remains only to prove Proposition 2.9. Recall the standard construction to compute $\kappa(x, y)$ uses a flow graph FG derived from the given graph $G = (V, E)$ by node splitting [Ahuja et al. 1993]. Specifically, FG has two vertices v_T, v_S and a unit capacity edge (v_T, v_S) for each $v \in V$. Furthermore, for G directed (undirected), each $(v, w) \in E$ gives rise to edge(s) (v_S, w_T) $((v_S, w_T)$ and $(w_S, v_T))$ in FG with infinite capacity. The source of FG is x_S and the sink is y_T.

For convenience, we restate the proposition. For directed graphs, δ denotes the smallest in- or out-degree.

PROPOSITION 2.9. *In a directed or undirected graph G, any quantity $\kappa(x, y)$ can be computed in the following time bounds:*

(*i*) $O(\sqrt{n}m/\log_{4n/\delta} n)$.

(*ii*) $O(n + (n - \delta)^{5/2}/\log(n - \delta))$, *assuming G is given as an adjacency matrix.*

PROOF

(*i*) Compute a max flow on FG using the algorithm of Feder and Motwani [1995]. It finds a maximum flow on a graph with unit vertex capacities in time

$$O(\sqrt{n}m/\log_b n)$$

for $b = n^2/m$. FG has $n(FG) = 2n$ and $m(FG) \geq \delta n$. For the latter note that if G is undirected then $m(FG) = 2m + n \geq \delta n$ and if G is directed then $m(FG) = m + n \geq \delta n$. So we always have $b(FG) \leq 4n/\delta$, giving (i).

(*ii*) The bound of Feder and Motwani [1995] displayed above is always $O(n^{2.5}/\log n)$. So for $\delta \leq n/2$ part (i) gives the bound of part (ii). Now assume $\delta > n/2$.

Replace FG by an equivalent graph CG on only $O(n-\delta)$ vertices, constructed as follows. Contract each edge directed from the source, (x_S, v_T), into a new source called S. This is valid since (x_S, v_T) has infinite capacity. It leaves $\leq n-\delta$ vertices of the type v_T. Similarly contract each edge directed to the sink, (v_S, y_T), into a new sink called T. This leaves $\leq n-\delta$ vertices v_S. (Note that the source x_S of FG remains part of the source in CG, and similarly for the sink y_T.)

CG is a unit vertex capacity graph: Each vertex v_T (v_S) remaining in CG still has exactly 1 outgoing (incoming) edge. (Note there may be many parallel unit capacity edges from S to T, corresponding to vertices $v \in V$ with $(x, v), (v, y) \in E$. This is fine.) Find a maximum flow on CG using Feder and Motwani [1995]. The opening remark shows this algorithm runs in time $O((n-\delta)^{5/2}/\log(n-\delta))$.

To complete the argument, we show that CG can be constructed in time $O(n + (n-\delta)^2)$. (To check this is within the desired time bound simply examine the cases $n-\delta \leq \sqrt{n}$ and the opposite.) Let

$$X = \{v : (x, v) \in E\}, \quad Y = \{v : (v, y) \in E\}.$$

The unit capacity edge (v_T, v_S) of FG becomes the following unit capacity edge in CG:

an edge (S, T) if $v \in X \cap Y$,
(S, v_S) if $v \in X - Y$,
(v_T, T) if $v \in Y - X$,
(v_T, v_S) if $v \notin X \cup Y$.

A (directed or undirected) edge (v, w) of E having $v \notin Y \cup \{x, y\}$ and $w \notin X \cup \{x, y\}$ gives, in CG, an infinite capacity edge (v_S, w_T). We can omit edges leaving T or entering S since they are irrelevant to a max flow. This justifies ignoring the possibilities $v \in Y \cup \{y\}$ and also $w \in X \cup \{x\}$.

We construct sets X and Y in $O(n)$ time. Given X and Y the unit capacity edges of CG are also constructed in $O(n)$ time. Given X, Y and the adjacency matrix for G, the infinite capacity edges of CG are constructed in $O((n-\delta)^2)$ time. $\square$

3. *Gap Enlargement*

Throughout this section, *gap* refers to the quantity $\delta - \kappa$ for the graph under discussion. This is a slight change from last section but should cause no confusion. In this section, we give an algorithm that modifies the given graph to enlarge its gap. This Gap Enlarger Algorithm sets up the Gap Algorithm of the previous section.

We start with a useful subproblem on domination. The directed version of this problem arises in the directed gap enlarging algorithm (Section 4) so we treat the directed version here as well.

Let G be a directed or undirected graph. A set of vertices D is a *complementary dominator* of a set of vertices S if for each $s \in S - D$, some vertex $d \in D$ is missing the edge (d, s). When G is undirected, this means s is not a neighbor of some $d \in D$. When G is directed, it means s is not an out-neighbor of d.

The undirected version of complementary domination easily reduces to the directed version: If G is undirected, let $\vec{G}$ be the directed version of G, that is, every edge of G gives rise to two antiparallel directed edges. Then, D is a complementary dominator of S in G exactly when it is in $\vec{G}$.

We give a simple greedy algorithm to find a small complementary dominator of an arbitrary set S. The algorithm is stated for directed graphs. If the graph is undirected simply replace each occurrence of "out-neighbor" by "neighbor".

Greedy Complementary Dominator Algorithm

Initialize the dominator set D to $\emptyset$. Repeat the following step until $S = \emptyset$: Choose a vertex y with as few out-neighbors in S as possible. Add y to D and delete from S every vertex that is not an out-neighbor of y. (This includes y, if it belongs to S.)

LEMMA 3.1. *Consider a directed or undirected graph G with a given set of vertices S. The above greedy algorithm finds a complementary dominator of S that has $O(1 + \log_{n/r}|S|)$ vertices. Here r is an upper bound on the in-degree (degree) of a vertex of S for G directed (undirected) respectively.*

Remark. If $r = 0$, the logarithmic term in the above bound is naturally interpreted as 0. Hence, in this case, the bound of the lemma is $O(1)$.

PROOF. It suffices to give the proof for a directed graph, by the above reduction.

Let s_i denote the number of vertices in S at the start of the ith iteration. (So s_1 is the cardinality of the input set S.) We claim that the vertex y chosen in the ith iteration has $\leq s_i r/n$ out-neighbors in S. In proof, the total number of edges directed to S is $\leq s_i r$. So some vertex is the tail of $\leq s_i r/n$ of these edges.

The claim implies $s_{i+1} \leq s_i r/n$, since after y is added to D every vertex remaining in S is an out-neighbor of y. This inequality implies the lemma: If $r = 0$, we get $s_2 = 0$, so $|D| = 1$. If $r > 0$, then every iteration decreases the quantity s_i by the factor n/r, again implying the bound of the lemma. $\square$

A time bound of $O(n + |D|m)$ for this algorithm is immediate. This suffices for our purposes. We remark that it is a simple exercise to implement the algorithm in linear time $O(m + n)$. One way is to use this data structure: The graph is stored in an adjacency list representation (both out-lists and in-lists) with all lists sorted (by vertex number). S is stored as a list, sorted by vertex number. Each vertex maintains a count of the number of its out-neighbors in S. Finally, for every i, $0 \leq i \leq |S|$, there is a doubly linked list of vertices whose count equals i. Further details of the linear-time implementation are left to the reader.

We turn to gap enlargement. The idea of the Gap Enlarger is that once we identify a vertex in a κ-separator we can delete it. This decreases κ by 1. Then, we can add appropriate edges to preserve δ. The result is a gap enlarged by 1.

Consider graphs $G = (V, E)$ of vertex connectivity κ and $G' = (V', E')$ of vertex connectivity κ', with $V' \subseteq V$. We say G' is *subconformal* to G when the following condition holds: (C^*, A^*, B^*) is a κ'-separation triple of G' if and only if $(C^* \cup (V - V'), A^*, B^*)$ is a κ-separation triple of G. In particular, $\kappa = \kappa' + |V - V'|$. When $V' = V$ (i.e., G and G' have the same κ-separation triples) we say G' is *conformal* to G.

As an example, observe that if S is a subset of every κ-separator of G, then the subgraph $G - S$ is subconformal to G. This is easy to verify since the hypothesis implies $\kappa(G - S) = \kappa - |S|$.

LEMMA 3.2

(i) *Let $R \subseteq V$ be 1-superrich. Any vertex x with $\kappa(x) = \kappa$ has $\kappa = \min\{\kappa(x, y) : y \in R\}$.*

(*ii*) *For any vertex x with* $\kappa(x) > \kappa$, $G - x$ *is subconformal to* G.

(*iii*) *For any vertices* x, y *with* $\kappa(x, y) > \kappa$, $G + (x, y)$ *is conformal to* G.

PROOF

(*i*) Some κ-separation triple (C^*, A^*, B^*) has $x \in A^*$. R contains a vertex $y \in B^*$. Thus, $\kappa(x, y) = \kappa$.

(*ii*) x belongs to every κ-separator of G. So (ii) follows from the observation preceding the lemma.

(*iii*) No κ-separation triple of G has x and y on opposite shores. Hence, any κ-separation triple of G is a κ-separation triple of $G + (x, y)$. The opposite inclusion is obvious. □

Now we present the gap enlargement algorithm. It is called with a graph G and parameter value Δ. We assume that G has $n > \delta + \Delta$ and $\delta, \Delta > 0$. Note these inequalities imply $n > \delta + 1$ so G is not complete and separators exist.

The purpose of the algorithm is to either find a κ-separator or increase the gap by Δ. More precisely the algorithm returns a separation triple (C, A, B) of G and a graph H, such that either $|C| = \kappa$ or H is subconformal to G and $\delta(H) - \kappa(H) = \delta(G) - \kappa(G) + \Delta$.

The algorithm constructs H by starting with G and repeatedly deleting vertices and adding edges. In the algorithm, variable H is maintained to be the current graph (the result of all deletions and additions). Initially, H is G.

Since H results from deleting vertices of G and adding edges, a separation triple (C, A, B) of H always gives a separation triple $(C \cup (V - V(H)), A, B)$ of G. Call this triple $(C \cup (V - V(H)), A, B)$ the *expansion* of (C, A, B). As before the algorithm maintains (C_0, A_0, B_0) as a separation triple of G with the smallest separator C_0 seen so far. So each time the algorithm finds a separation triple of H, we use its expansion to update (C_0, A_0, B_0).

The algorithm maintains the following two invariants:

I1. If κ has not been found then H is subconformal to G.

I2. At the start of each iteration of the main loop (i.e., when Step (1) is reached) vertex a has degree exactly δ.

As before we present the algorithm along with a verification of most of the details for correctness. Specifically we verify that each step of the algorithm is well defined, and Invariants I1–I2 are always preserved.

Gap Enlarger Algorithm

In this algorithm δ always denotes $\delta(G)$. Let a be a vertex of degree δ. Initialize (C_0, A_0, B_0) to the separation triple that has C_0 consisting of the neighbors of a. Initialize H to G. Repeat the following two steps Δ times:

Step (1). Choose a vertex x of $H - a$. Find $\kappa(x; H)$ and a corresponding separation triple. Use the triple's expansion to update (C_0, A_0, B_0). Delete x from H.

Explanation. If we have not yet found κ, then $\kappa(x; H) > \kappa(H)$. Hence, Lemma 3.2(ii) applies to show Invariant I1 is preserved.

Step (2). Let S be the set of vertices of degree $<\delta$. (Each vertex of S was adjacent to x and now has degree $\delta - 1$.) Let D be a complementary dominator of S in graph H. Do Step (2.1) for each vertex $y \in D$:

Step (2.1). Find $\kappa(y; H)$ and a corresponding separation triple. Use the triple's expansion to update (C_0, A_0, B_0). Let Z be the set of all vertices in S that are not adjacent to y. Remove each vertex of Z from S.

Step (2.1.1). If $Z \neq \{y\}$, then do the following: For each $z \in Z - y$, add edge (y, z) to H. Set a to an arbitrary vertex of Z.

Step (2.1.2). If $Z = \{y\}$, then do the following: Choose any vertex x that is not adjacent to y and add edge (x, y) to H. Set a to y.

Explanation. There are three points: (a) If we have not yet found κ then $\kappa(y; H) > \kappa(H)$ and Lemma 3.2(iii) applies. Thus, each time Steps (2.1.1)–(2.1.2) add an edge Invariant I1 is preserved. (b) Invariant I2 holds since Steps (2.1.1)–(2.1.2) set a to a vertex whose degree has just increased from $\delta - 1$ to δ. (c) In Step (2.1.2), the desired vertex x exists, since y has degree $\delta - 1$ and H has $\geq n - \Delta > \delta$ vertices.

The next lemma summarizes the correctness properties of the algorithm, and also gives the time bound. It uses the value

$$\beta = \frac{n - \Delta}{\delta}.$$

LEMMA 3.3. *Assume the Gap Enlarger is called with $n > \delta + \Delta$ and $\delta, \Delta > 0$.*

(*i*) *The final graph H has $n(H) = n - \Delta$, $m(H) \leq m$, $\delta(H) = \delta$. Furthermore, the expansion of any separation triple of H is a separation triple of G.*

(*ii*) *Either the Gap Enlarger finds κ, or H has gap $\delta(H) - \kappa(H) \geq \Delta$ and is subconformal to G. (In particular, $\kappa(H) = \kappa(G) - \Delta$.)*

(*iii*) *The time is $O(\Delta n \min\{m,\ n + (n - \delta)^2\} \log_\beta n)$.*

PROOF

(i) –(ii) Most details of (i) – (ii) are clear so we just check three points:

$m(H) \leq m$ holds since Steps (2.1.1)–(2.1.2) add an edge incident to a vertex z only if Step (1) has deleted such an edge.

$\delta(H) = \delta$ holds because any vertex whose degree falls below δ in Step (1) has its degree restored to δ in Steps (2.1.1)–(2.1.2). Furthermore the existence of vertex a shows the minimum degree in H is δ and no more.

If the algorithm completes without finding κ the final graph H has gap $\delta(H) - \kappa(H) = \delta - (\kappa - \Delta) \geq \Delta$.

(iii) First observe that in each execution of Step (2) the dominator has size $|D| = O(\log_\beta n)$. This follows from the bound of Lemma 3.1, since $n(H) \geq n - \Delta$ and $r = \delta - 1 < \delta$ imply $n(H)/r \geq \beta$. (Also $n > \beta$ ensures $\log_\beta n = \Omega(1)$.)

Consider the time for Step (2) (excluding Step (2.1) and its substeps). It is dominated by the time for the Greedy Complementary Dominator Algorithm. Recall our simple time bound for this algorithm, $O(n + |D|m)$; since $\delta > 0$ this is $O(|D|m)$. Step (2) is executed Δ times, so the total time for it is $O(\Delta m \log_\beta n)$. This quantity is within the desired time bound (since $m = O(n^2)$).

Next consider the computation of rooted connectivity quantities $\kappa(x)$ in Steps (1) and (2.1). The above bound on $|D|$ shows a total of $O(\Delta \log_\beta n)$ such computations are done.

As mentioned in Section 1, one quantity $\kappa(x)$ can be computed in time $O(nm)$. This gives total time $O(\Delta nm \log_\beta n)$, corresponding to the first possibility for the time bound of (iii). In fact, it is easy to see that this bound also accounts for the remaining time for the algorithm (the vertex deletions in Step (1) and the edge additions in Steps (2.1.1)–(2.1.2)).

The second possibility for the time bound is based on the fact that $\kappa(x)$ can be computed in time $O(n(n-\delta)^2)$ (even for digraphs). This follows from Henzinger et al. [2000, p. 233]. There the bound $O(n(n-\kappa)^2)$ is shown but the same argument proves the stronger bound. (The argument of Henzinger et al. [2000] uses the same idea as Proposition 2.9(ii). It only relies on the fact that the degree of x is $\geq\kappa$. Since the degree of x is also $\geq\delta$ the claimed bound follows.) The bound assumes that the graph is represented as an adjacency matrix, but this presents no problem (since the second time bound of (iii) allows time $\Omega(n^2)$ for each connectivity computation).

Thus, we get total time $O(\Delta n(n-\delta)^2 \log_\beta n)$ for rooted connectivity computations. It is trivial to see that remaining time for vertex deletions and edge additions is $O(\Delta n^2 \log_\beta n)$. Hence, the second possible time bound of (iii) holds. $\square$

We remark that the alternate bound for rooted connectivity $\kappa(x)$ just mentioned need not hold for a quantity $\kappa_W(S)$. But we do not require it. However, for use in Section 6 we note that Henzinger et al. [2000] shows κ can be computed in

$$O(\,(n(n-\delta))^2\,)$$

time (even for digraphs). Again, the bound $O(\,(n(n-\kappa))^2\,)$ is proved in Henzinger et al. [2000, p. 233], and for the same reasons as above κ can be replaced by δ in this proved bound.

When $\delta \leq \sqrt{n}$, flow computations become cheaper. This makes an alternate implementation of the Gap Enlarger faster. We call this implementation the Flow Version. We first define a notion that is used in this and other flow-oriented algorithms (Section 5). We require a *conditional computation* of $\kappa(x)$ to return the correct value $\kappa(x)$ along with a corresponding separator if κ has not been found and furthermore $\kappa(x) = \kappa$. Otherwise (if κ has been found or if $\kappa(x) > \kappa$), the computation can return any valid separator and corresponding value, or even the value $n-1$ with no separator (in both cases, the value returned will be $\geq\kappa$). It is immediate that the Gap Enlarger remains correct if it uses conditional rooted connectivity computations.

The *Flow Version of the Gap Enlarger* is simply the basic Gap Enlarger modified to use conditional computations of $\kappa(x)$. We specify the Flow Version by just giving the changes made to the above Gap Enlarger Algorithm:

Before Step (1), find $\kappa(a)$ and a corresponding separation triple. Initialize (C_0, A_0, B_0) to this triple. Initialize R to the set of neighbors of a. If κ has not yet been found, then Lemma 2.2(i) shows R is 1-superrich.

In Step (1), change the computation of $\kappa(x; H)$ to a conditional computation. Do this by computing the values $\kappa(x, y; H)$, $y \in R$, as specified in Lemma 3.2(i). In Step (2.1), compute $\kappa(y; H)$ conditionally, in the same way.

Whenever Step (1) deletes a vertex x from H, delete it from R also. Note that R remains 1-superrich if we have not yet found κ (since x is not in any κ-shore). Hence, each conditional rooted connectivity computation works correctly.

LEMMA 3.4. *Assume the Flow Version of the Gap Enlarger is called with $n \geq 2\Delta$, $\sqrt{n} \geq \delta > 2$ and $\Delta > 0$.*

(i) *Lemma* 3.3(i)–(ii) *continue to hold.*
(ii) *The time is* $O((n+\delta^2\Delta)m)$.

PROOF

(i) The assumption $n > \delta + \Delta$ of Lemma 3.3 holds since the lemma's hypothesis implies $\delta+\Delta \le \sqrt{n}+n/2 < n$. (The last inequality follows from the hypothesis $\sqrt{n} > 2$.) The proof of Lemma 3.3(i)–(ii) holds without modification.

(ii) First observe that in each execution of Step (2) the dominator has size $O(1)$. In proof Lemma 3.3 shows the size is $O(\log_\beta n)$. This quantity is $O(1)$ since $\beta = (n-\Delta)/\delta \ge (n/2)/\sqrt{n} = \sqrt{n}/2$.

Since Step (2) is executed $O(\Delta)$ times, it uses total time $O(\Delta|D|m) = O(\Delta m)$. We also get that there are a total of $O(\Delta)$ conditional rooted connectivity computations in Steps (1) and (2.1). Each such computation consists of $|R| \le \delta$ max flow computations. Furthermore each max flow computation takes time $O(\delta m)$, since we can stop when a flow of value δ has been found. (Recall from Section 1 that we always have $\delta \ge \kappa$.) So the total time for max flow computations in Steps (1) and (2.1) is $O(\Delta\delta^2 m)$. It is easy to see that this also accounts for the time modifying the graph in Steps (1) and (2.1.1)–(2.1.2).

The final contribution to the time is the computation of $\kappa(a)$ before Step (1). As previously noted, this is $O(nm)$ [Henzinger et al. 2000]. The time bound of (ii) follows. $\square$

The overall connectivity algorithm for undirected graphs is essentially complete–we just need to select an appropriate value of Δ. (Note that creating a larger gap takes more time, but a larger gap allows faster connectivity checking.) So readers only interested in the undirected algorithm can skip to Section 6.

4. *Digraphs and Gap Enlargement*

This section reviews our terminology for digraphs and presents the results corresponding to Section 3 for gap enlargement on digraphs.

These basic terms from Section 1 are defined in exactly the same way: separation triple, separator, shore, vertex connectivity κ, κ-separator, κ-shore. Note that a separation triple (C, A, B) has no edge from A to B (by definition) but can have edges from B to A. The value $\kappa(x, y)$ is also defined in the same way. For digraphs, $\kappa(y, x)$ can differ from $\kappa(x, y)$.

For other terms, we use superscripts $^-$ and $^+$ to refer to incoming and outgoing edges, respectively. For instance, δ^+ is the smallest out-degree of a vertex, $\kappa^-(x) = \min\{\kappa(y, x) : y \in V\}$, etc. Define

$$\delta = \min\{\delta^+, \delta^-\}, \quad \kappa(x) = \min\{\kappa^+(x), \kappa^-(x)\}, \quad \kappa_W(S) = \min\{\kappa_W^+(S), \kappa_W^-(S)\}.$$

If (C, A, B) is a κ-separation triple, A is a κ^+-*shore* and B is a κ^--*shore*.

An *out-neighbor* (*in-neighbor*) of a set of vertices X is a vertex $y \notin X$ on an edge (x, y) ((y, x)) for some $x \in X$, respectively.

The time bounds for computing $\kappa(x, y)$ and $\kappa(x)$ given in Section 1 remain valid for digraphs.

We now present the results in Sections 2–3 that have close directed analogs. We begin with Lemma 2.2. The notions of rich and superrich are defined as before. An analog of part (i) holds but is not useful, so we only generalize (ii).

LEMMA 4.1. *Any set $S \subseteq V$ with $\kappa_W(S) > \kappa$ is ρ-superrich for*

$$\rho = \min\{\kappa_W(S),\ \delta + 1\} - \kappa.$$

Remark. The lemma does not assume $\kappa_W(S)$ corresponds to a separation triple.

PROOF. A complete digraph has $\kappa = n - 1$. This precludes the existence of S, so we can assume the graph is not complete.

Let (C^*, A^*, B^*) be any κ-separation triple and assume $|A^* \cap S| < \rho$. Since every vertex has out-degree at least δ, $|A^*| + |C^*| > \delta$. Thus $|A^*| \geq \delta - \kappa + 1 \geq \rho$, which implies $A^* - S \neq \emptyset$. This makes $(C^* \cup (A^* \cap S), A^* - S, B^*)$ a separation triple. Hence, $\kappa_W^-(S) \leq |C^* \cup (A^* \cap S)| < \kappa + \rho \leq \kappa_W(S)$, a contradiction. □

The Expander Lemma 2.8 applies to digraphs without change. Note that in the formula for κ each edge of $\chi(R, d)$ gives rise to two quantities $\kappa(x, y)$.

We turn to the directed version of Lemma 3.2. This lemma will be applied in both the Gap Enlarger of this section and the Gap Algorithm of the next section. The new lemma also contains some related facts that are used in the Gap Algorithm.

We first extend the notation for weak separation κ_W as follows. For $S \subseteq V$ and $x \in V$, define

$$\kappa_W(S, x) = \min\{|S|,\ n - 1,\ |C| \ :\ (C, A, B) \text{ is a separation triple of } G \text{ that has } S \subseteq A \cup C,\ x \in B\}.$$

If $x \in S$, then no separators exist and $\kappa_W(S, x) = \min\{|S|, n - 1\}$. To compute $\kappa_W(S, x)$ when $x \notin S$ first form the graph $G_W(S)$ as described in Section 1 (add a vertex s to G along with directed edges (s, v), $v \in S$). Then, $\kappa_W(S, x) = \kappa(s, x; G_W(S))$. $\kappa_W(x, S)$ is defined and computed analogously.

The definition of conformal and subconformal is unchanged from Section 3. Parts (i)–(iii) of the following lemma are exact analogs of Lemma 3.2 and the proofs go over unchanged. Throughout our discussion of digraphs we state results explicitly for one direction $^+$ or $^-$ (e.g., part (i), below). The other direction always holds by considering the reverse graph.

LEMMA 4.2

(*i*) *Let $R \subseteq V$ be 1-superrich. Any vertex x with $\kappa^+(x) = \kappa$ has $\kappa = \min\{\kappa(x, y) : y \in R\}$. Similarly, any set of vertices S with $\kappa_W^+(S) = \kappa$ and $|S| > \kappa$ has $\kappa = \min\{\kappa_W(S, y) : y \in R\}$.*

(*ii*) *For any vertex x with $\kappa(x) > \kappa$, $G - x$ is subconformal to G.*

(*iii*) *For any vertices x, y with $\kappa(x, y) > \kappa$, $G + (x, y)$ is conformal to G.*

(*iv*) *For any vertex x and any set of vertices S with $\kappa^+(x) < \kappa_W^+(S)$, $\kappa^+(x) = \min\{\kappa(x, y) : y \in S\}$.*

(*v*) *For any set of vertices S, any separation triple (C, A, B) and any vertex $b \in B - S$, $\kappa_W(S, b) \leq |C| + |S \cap B|$.*

PROOF

(i) Note that in the second assertion of this part, the hypothesis $|S| > \kappa$ is actually unnecessary: If $|S| = \kappa$, then we always have $\kappa_W^+(S) = \kappa_W(S, y) = \kappa$, but for trivial reasons. We include the hypothesis to ensure that the lemma is used in a meaningful way.

(iv) Let (C, A, B) be a separation triple corresponding to $\kappa^+(x)$. If B and S are disjoint, then $S \subseteq A \cup C$ and so $\kappa^+_W(S) \leq |C| = \kappa^+(x)$, a contradiction. Thus, $B \cap S \neq \emptyset$, which gives (iv).

(v) The hypothesis makes $(C \cup (S \cap B), A, B - S)$ a separation triple included in the set defining $\kappa_W(S, b)$. This implies (v). □

Now we show that the algorithms of Section 3 for gap enlargement extend naturally to digraphs, maintaining the same time bound. The gap is still defined to be $\delta - \kappa$.

The Gap Enlarger Algorithm need only be modified to interpret things in a directed way. We will step through the algorithm noting the simple changes. Invariant I2 states that a has in- or out-degree δ.

Before Step (1) vertex a is chosen to have in- or out-degree δ. The separation triple (C_0, A_0, B_0) is chosen correspondingly, so as before $|C_0| = \delta$.

Step (1) is unchanged. It has the analogous explanation, Lemma 4.2(ii).

Step (2) must be executed twice, first to repair the vertices of in-degree $\delta - 1$ and then to repair those of out-degree $\delta - 1$. We will discuss the former, the latter being symmetric.

S is defined as the set of vertices of in-degree $<\delta$, and D its complementary dominator. Z is the set of all vertices in S that are not out-neighbors of y. There is just one change in Steps (2.1.1)–(2.1.2): Step (2.1.2) is executed whenever $y \in Z$. This is natural, since in this case we must increase the in-degree of y from $\delta - 1$ to δ. The explanation of Step (2) is as before, now using Lemma 4.2(iii).

COROLLARY 4.3. *The Gap Enlarger Algorithm as modified for digraphs satisfies Lemma* 3.3.

PROOF. The argument of Lemma 3.3 applies unchanged. □

We turn to the Flow Version of the Gap Enlarger. Assume for the moment that R gets initialized to a 1-superrich set as before. The rest of the algorithm is unchanged from Section 3: Steps (1) and (2.1) do conditional computations of $\kappa(x; H)$ and $\kappa(y; H)$ respectively, using Lemma 4.2(i).

So the only change needed in the Flow Version is the initialization before Step (1), in particular to make R 1-superrich. The complete procedure before Step (1) is as follows:

Let a be a vertex with in- or out-degree δ. Let R be a set of $\delta + 1$ vertices, specifically vertex a and its δ in- or out-neighbors. Find $\kappa_W(R)$ and a corresponding separation triple. Initialize (C_0, A_0, B_0) to this triple, and initialize H to G.

To justify this initialization, first note that by definition $R - a$ is a separator of δ vertices. Hence $\kappa_W(R) \leq \delta < |R|$ and (C_0, A_0, B_0) is initialized to a true separation triple.

Next suppose that (C_0, A_0, B_0) is not a κ-separator. Then, Lemma 4.1 shows that R is 1-superrich, as desired.

COROLLARY 4.4. *The Flow Version of the Gap Enlarger Algorithm as modified for digraphs satisfies Lemma* 3.4.

PROOF. There are just two trivial changes from the proof of the lemma. Now R contains $\delta + 1$ vertices rather than δ, but we still have $|R| = O(\delta)$. The initialization involves computation of a weak connectivity quantity $\kappa_W(R)$ rather than $\kappa(a)$, but the time remains $O(nm)$ as cited in Section 1. $\square$

5. *Digraph Gap Algorithm*

The nesting property for directed separation triples is weaker than its undirected counterpart Lemma 2.4. This leads to a more involved directed Gap Algorithm. In addition, the directed Gap Algorithm needs to be more general—it must handle low connectivities, since Henzinger's algorithm [Henzinger 1997] for low connectivities applies only to undirected graphs. Nonetheless the same asymptotic running time can be achieved for the directed Gap Algorithm.

Section 5.1 begins with our directed nesting property. Then two versions of the Gap Algorithm are presented: Section 5.2 gives an algorithm that is more efficient for high connectivities, and Section 5.3 gives the algorithm for low connectivities.

5.1. NESTING PROPERTY. We start by considering two arbitrary separation triples (C, A, B) and (C', A', B') as in Figure 1. The quadrants $A \cap A'$ and $B \cap B'$ have properties similar to the undirected case. We state these properties for both $A \cap A'$ and $B \cap B'$, for ease of future reference. We only prove the properties for $A \cap A'$, since reversing the graph shows the symmetric properties for $B \cap B'$.

(a) The out-neighbors of $A \cap A'$ are contained in $(A' \cap C) \cup (C' - B)$. The in-neighbors of $B \cap B'$ are contained in $(B' \cap C) \cup (C' - A)$.
(b) If $A \cap A'$ is nonempty, then its out-neighbors form a separator. If $B \cap B'$ is nonempty, then its in-neighbors form a separator.

To show (a) for $A \cap A'$, observe that no edge is directed from A to B, and similarly no edge is directed from A' to B'. Hence, an out-neighbor x of $A \cap A'$ does not belong to $(A \cap A') \cup B \cup B'$. So $x \in A'$ implies $x \in A' \cap C$ and $x \notin A'$ implies $x \in C' - B$. Now (b) for $A \cap A'$ follows since no vertex of B is an out-neighbor of $A \cap A'$.

In a separation triple (C, A, B), B is *in-extreme* if each of its nonempty subsets has at least $|C|$ in-neighbors; similarly for A *out-extreme*. We have the same examples as in Section 2, specifically the shore B of separation triple (C, A, B) is in-extreme if (C, A, B) is either

a κ-separation triple, or
a separation triple corresponding to $\kappa^+(x)$, more precisely $x \in A$ and $|C| = \kappa^+(x)$, or
a separation triple corresponding to $\kappa^+_W(S)$, more precisely $S \subseteq A \cup C$ and $|C| = \kappa^+_W(S)$.

The directed nesting lemma is illustrated in Figure 3. (As in Figure 2(a), in Figure 3(a), it is easy to see that $B \cap C^* = \emptyset$ always holds but we do not use this property.)

LEMMA 5.1 (DIRECTED NESTING LEMMA). *Let (C, A, B) be a separation triple with B in-extreme. Either*

(i) A contains a κ^+-shore, or

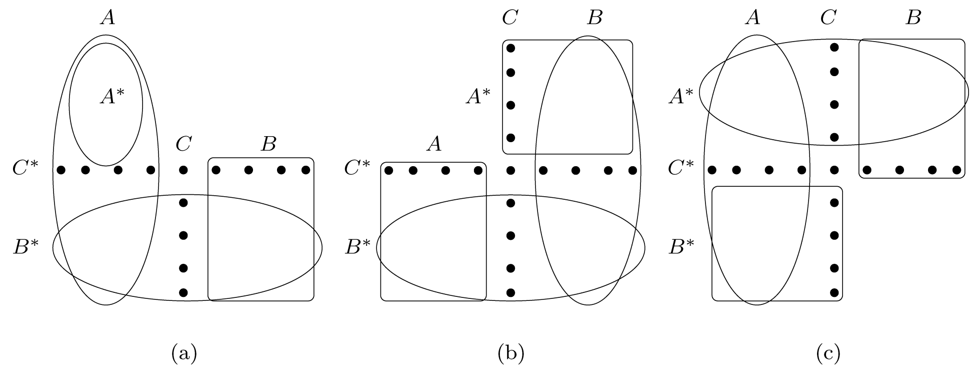

FIG. 3. The possible relations given by the Directed Nesting Lemma 5.1. (C, A, B) is a separation triple with B in-extreme. (C^*, A^*, B^*) is a κ-separation triple as in the lemma. The lemma's part (i) is illustrated in (a) and part (ii) is illustrated in (b)–(c).

(*ii*) *for every κ-separation triple (C^*, A^*, B^*) either $A \cap A^*$ or $B \cap B^*$ is empty; furthermore*

$A \cap A^* \neq \emptyset$ implies $|(B \cup C) \cap A^*| \geq |B|$, and symmetrically,
$B \cap B^* \neq \emptyset$ implies $|(A \cup C) \cap B^*| \geq |A|$.

PROOF. The argument divides into two cases depending on whether or not the first part of (ii) is satisfied. We first assume it is not.

Case (1). Some κ-separation triple (C^*, A^*, B^*) has $A \cap A^*,\ B \cap B^* \neq \emptyset$.

We will show that (i) holds, specifically, $A \cap A^*$ is a κ^+-shore. Property (b) and the assumption $A \cap A^* \neq \emptyset$ shows it suffices to prove $A \cap A^*$ has κ out-neighbors. We claim

$$|C^* \cap B| \geq |C \cap A^*|.$$

This claim implies the desired conclusion: Using property (a) and the claim, the number of out-neighbors of $A \cap A^*$ is at most $|A^* \cap C| + |C^* - B| \leq |C^* \cap B| + |C^* - B| = |C^*|$.

Now we prove the claim. The assumption $B \cap B^* \neq \emptyset$ implies $B \cap B^*$ has at least $|C|$ in-neighbors, by in-extremeness of B. So property (a) for $B \cap B^*$ gives $|B^* \cap C| + |C^* - A| \geq |C|$. Rearrange this to get $|C^* - A| \geq |C - B^*|$. Since the complement of A is $B \cup C$ and the complement of B^* is $A^* \cup C^*$, the inequality becomes $|C^* \cap B| + |C^* \cap C| \geq |C \cap A^*| + |C \cap C^*|$. This is equivalent to the claim.

Case (2). Every κ-separation triple (C^*, A^*, B^*) has $A \cap A^*$ or $B \cap B^*$ empty.

The assumption is the first part of (ii) so we need only prove the second part. The argument will not use the extremeness of B, that is, it is symmetric in A and B. So, without loss of generality, assume $A \cap A^* \neq \emptyset$ and $B \cap B^* = \emptyset$.

Properties (a) and (b) for $A \cap A^*$ show $|A^* \cap C| + |C^* - B| \geq |C^*|$, or equivalently $|A^* \cap C| \geq |C^* \cap B|$. Thus

$$|(B \cup C) \cap A^*| = |B \cap A^*| + |C \cap A^*| \geq |B \cap A^*| + |C^* \cap B| = |B|. \qquad \square$$

5.2. GAP ALGORITHM FOR HIGH CONNECTIVITIES. We turn to the directed Gap Algorithm. As in the undirected case, the algorithm checks k-connectedness for a given k. Similarly, the algorithm maintains (C_0, A_0, B_0) as a triple having the smallest separator C_0 seen so far. Define

$$D = \min\{2k, \delta\}, \quad \gamma = D - k, \quad \tau = \gamma/2.$$

The Gap Algorithm assumes $\gamma \geq 4$. No other assumptions on the input are made. For future reference, we note some simple inequalities satisfied by the parameters:

$$\delta \geq D, \quad D = k + 2\tau.$$

As before, we explain each step as it is presented. These explanations collectively verify the correctness of the algorithm. Specifically, they show that if $\kappa < k$ the algorithm finds a κ-separator. So the explanations always assume $\kappa < k$.

The algorithm modifies the given graph $G = (V, E)$ to a new graph H on V. Initially H is G, and H always denotes the current graph (the result of all modifications). H is constructed by adding edges to G, so any separation triple of H is a separation triple of G.

The algorithm maintains two sets of vertices X and Y. Intuitively, X avoids any κ^--shore, so its vertices belong to either the $^+$-shore or the separator of any κ-separation triple. Y is symmetric, it avoids any κ^+-shore. More precisely, the algorithm maintains these invariants:

> *I1*. If κ has not been found then H is conformal to G and for every κ-separation triple (C^*, A^*, B^*), $X \cap B^* = Y \cap A^* = \emptyset$.
>
> *I2*. After each iteration of Step (3), every vertex of X (Y) has in-degree (out-degree) $n - 1$, respectively.

The algorithm does its connectivity computations in H. It also finds rich sets in H. Conformality (Invariant I1) ensures the results are valid: If the algorithm returns κ and a κ-separator, conformality implies $\kappa(G) = \kappa(H)$ and the κ-separation triple found in H is valid in G. Similarly, a set is rich in G if and only if it's rich in H.

The overall outline shares features with the undirected Gap Algorithm: Steps (1)–(3) either find κ or find a τ-rich set R, which is processed in Step (4). In more detail, assuming $\kappa < k$, Step (3) applies the Directed Nesting Lemma 5.1 to either find κ or to make progress by adding edges that don't change the κ-separators (i.e., they maintain conformality). Eventually, Step (3) either finds κ or finds a τ-rich set R by using Lemma 4.1. In the latter case, Step (4) finds κ using the Expander Lemma 2.8 as in the undirected algorithm.

Directed Gap Algorithm, Basic Version

Step (1). If $n = \delta + 1$, then the graph is k-connected so return. If $n < 2D$, then set R to V and go to Step (4).

Explanation. As in the undirected algorithm, if $n = \delta + 1$ the graph is complete and $\kappa = \delta \geq D = k + 2\tau > k$. After this test, we can assume the graph is not complete, so separators exist. Now suppose $n < 2D$. As in the undirected case, Step (4) requires that R be τ-rich. If $\kappa < k$, then V is τ-rich even without the inequality $n < 2\delta$: Any κ-shore has $\geq \delta + 1 - \kappa > \delta - k \geq D - k = 2\tau$ vertices.

Step (2). Let S be an arbitrary set of $2D$ vertices. Initialize (C_0, A_0, B_0) to a separation triple that corresponds to a vertex of in- or out-degree δ, so $|C_0| = \delta$. Initialize graph H to G and vertex sets X and Y to $\emptyset$.

Step (3). Repeat Steps (3.1)–(3.2) until Step (3.2) either halts or goes on to Step (4): /$*$ Steps (3.1)–(3.2) do not change the set S, but they do change the graph H and sets X, Y. $*$/

Step (3.1). Find $\kappa_W(S; H)$ and a corresponding separation triple (C, A, B) if one exists. In the latter case update (C_0, A_0, B_0).

Step (3.2). /$*$ Although S never changes the quantity $\kappa_W(S; H)$ computed in Step (3.1) can change because of edges added to H. Throughout Step (3.2) (C, A, B) denotes the separation triple corresponding to the current $\kappa_W(S; H)$, if it exists. If $\kappa < k$ has not been found, Step (3.2) either finds it or applies Lemma 4.1 or 5.1 to either find a τ-rich set R or add edges to H preserving conformality (Invariant I1). $*$/

Step (3.2.1). If $\kappa_W(S; H) \geq k + \tau$, then let R be the set S and go to Step (4).

Explanation. Assuming $\kappa < k$, R is a τ-rich set of $2D$ vertices. To show richness observe that $\kappa_W(S; H) \geq k + \tau > \kappa$ shows we can apply Lemma 4.1 to S. Since the algorithm never deletes edges, $\delta(H) \geq \delta \geq D > k + \tau$. Now the lemma shows S is τ-rich since $\rho = \min\{\kappa_W(S; H), \delta(H) + 1\} - \kappa > (k + \tau) - k = \tau$.

Step (3.2.2). Consider the opposite case $\kappa_W(S; H) < k + \tau$. /$*$ This inequality shows $\kappa_W(S; H)$ does in fact correspond to a separation triple (C, A, B), since $|S| = 2D > D > k + \tau$. Step (3.2.2.1) treats the case $\kappa_W(S; H) = \kappa_W^+(S; H)$ and Step (3.2.2.2) treats the symmetric case $\kappa_W(S; H) = \kappa_W^-(S; H)$. $*$/

Step (3.2.2.1). Suppose $\kappa_W(S; H) = \kappa_W^+(S; H)$. Thus, $S \subseteq A \cup C$ and B is in-extreme. This shows we can apply the Directed Nesting Lemma 5.1 to (C, A, B). We do so as follows:

Compute $\kappa_W(A; H)$ and $\kappa_W(C; H)$ with corresponding separation triples and update (C_0, A_0, B_0). Add B to X. If this makes $|X| > k$, then compute $\kappa_W(X; H)$ and a corresponding separation triple, update (C_0, A_0, B_0) and halt. Otherwise, add edge (x, y) to H for every $x \in V$, $y \in B$ where this edge does not exist.

Explanation. First, note that Invariant I1 becomes vacuous if we find κ. Invariant I2 is obviously preserved by the new edges.

As noted above, Lemma 5.1 applies to (C, A, B) and B. So one of the following three possibilities (a)–(c) must hold:

(a) A contains a κ^+-shore: We find κ since $\kappa = \kappa_W^-(C; H) = \kappa_W(C; H)$. (We must have $|C| \geq \kappa$ for this equation to hold, and in fact we need $|C| > \kappa$ to guarantee that a κ-separator is found. We can assume $|C| > \kappa$ since otherwise κ has been found.)

(b) Some κ-separation triple (C^*, A^*, B^*) has $A \cap A^* = \emptyset$: Again we find κ since $\kappa = \kappa_W^-(A; H) = \kappa_W(A; H)$. (As in (a) this requires $|A| > \kappa$. This holds since $|A| \geq |S| - |C| > 2D - (k + \tau) > D > \kappa$.)

(c) Every κ-separation triple (C^*, A^*, B^*) has $A \cap A^* \neq \emptyset$: Lemma 5.1(ii) shows $B \cap B^* = \emptyset$ for every (C^*, A^*, B^*). Adding B to X preserves the equation of Invariant I1. If $|X| > k$, then I1 shows we find κ as $\kappa_W^+(X; H) = \kappa_W(X; H)$. (This requires our assumption $k > \kappa$.) If $|X| \leq k$ then we add new edges directed into vertices of B, and Lemma 4.2(iii) shows invariant I1 is preserved. (Use $B \cap B^* = \emptyset$ here.)

Step (3.2.2.2). Suppose $\kappa_W(S; H) = \kappa^-_W(S; H)$. Thus $S \subseteq B \cup C$ and A is in-extreme. This step is symmetric to Step (3.2.2.1), for example, it adds A to Y and adds edges directed from the vertices of A. After adding their edges, both Steps (3.2.2.1) and (3.2.2.2) go back to Step (3.1) to start the next repetition.

Explanation. It is not entirely clear why Step (3) cannot loop indefinitely. This is proved as part of the efficiency analysis in (a) below.

Step (4). /* If κ has not been found and $\kappa < k$, then R is now τ-rich and either $R = V$ with $n < 2D$ (from Step (1)) or $|R| = 2D$ (from Step (3.2.1)). */

Apply the Expander Lemma 2.8 with $\rho = \tau$, constructing the graph $\chi(R, d)$ and computing the values $\kappa(x, y; G)$ specified in the lemma (for digraphs). Update (C_0, A_0, B_0) for these values.

Explanation. If Step (4) starts with $\kappa < k$ and κ still not found, the Expander Lemma guarantees that Step (4) computes κ.

We have verified that the algorithm is correct, as long as it halts (i.e., Step (3) does not loop indefinitely). Now we analyze the algorithm's efficiency (and show that it halts). We begin by reviewing the time for the four steps. Steps (1)–(2) are $O(m)$. We claim Step (3) uses total time

$$O(knm/\gamma).$$

We establish this by showing

(a) the loop of Step (3) executes Steps (3.1)–(3.2) $O(k/\tau)$ times, and
(b) each execution of Steps (3.1)–(3.2) uses time $O(nm)$.

Since $\tau = \gamma/2$ these two facts imply the claimed bound. (Note that (a) proves the algorithm halts.)

PROOF OF (a). First observe $k \geq \tau$. This follows since the definition of our quantities shows $2k \geq D \geq 2\tau$. Hence, the desired bound $O(k/\tau)$ certainly allows $O(1)$ executions.

To bound the number of executions, note that an execution of Step (3.2) is the last one if either $\kappa_W(S; H) \geq k + \tau$ (Step (3.2.1)) or, in the opposite case, X or Y is enlarged to $>k$ vertices (Step (3.2.2.1)–(3.2.2.2)). We claim that each execution with $\kappa_W(S; H) < k + \tau$ increases $|X| + |Y|$ by at least τ. Clearly the claim implies there are $\leq 2k/\tau$ such executions, yielding the bound of (a).

Assume $\kappa_W(S; H) = \kappa^+_W(S; H)$, the case $\kappa_W(S; H) = \kappa^-_W(S; H)$ being symmetric. Invariant I2 shows that any separation triple (C', A', B') has $X \cap B' = \emptyset$. Applying this to (C, A, B), the triple corresponding to $\kappa^+_W(S; H)$ in Step 3.1, gives $X \cap B = \emptyset$. We are assuming that $|C| = \kappa^+_W(S; H) < k + \tau$. This plus the fact that every degree is $\geq \delta \geq D = k + \gamma$ gives

$$|B| \geq \delta + 1 - |C| > (k + \gamma) - (k + \tau) = \tau.$$

Thus, when Step (3.2.2.1) adds B to X, $|X|$ increases by $\geq \tau$ as claimed. $\square$

PROOF OF (b). First observe that we always have $m(H) = O(m)$. In proof, Step (3.2.2) only adds edges when $|X| \leq k$ for the newly enlarged set X, or similarly $|Y| \leq k$ for the new Y. Hence, $O(kn)$ edges are added to H. This quantity is $O(m)$ since $k \leq \delta$ and $\delta n \leq m$.

It is easy to see that the total time for adding edges in all executions of Step (3.2.2) is $O(m)$, since the bookkeeping is trivial.

The remaining time in one execution of Steps (3.1)–(3.2) is dominated by $O(1)$ rooted connectivity computations $\kappa(x)$ or $\kappa_W(S)$ in graph H. As mentioned in Section 1 each such computation uses time $O(n(H)m(H)) = O(nm)$, giving (b). $\square$

We characterize the time for Step (4) exactly the same as in the undirected case: Step (4) constructs $\chi(R, d)$ and performs $2m(\chi(R, d))$ max flow computations of quantities $\kappa(x, y)$. Let $r = |R|$. From (4) with $\rho = \tau = \Theta(\gamma)$ we get the time amounts to

$$O(r^3/\gamma(r - k))$$

computations of $\kappa(x, y)$.

Now we give four time bounds for the algorithm, similar to Lemma 2.10. As before (ii)–(iii) are for the extreme case of κ and δ very close to n (Lemma 6.2). These two bounds assume the algorithm computes max flows using Proposition 2.9. Note also that the definition of D shows we can replace it by k or δ in all the time bounds (e.g., (i) becomes $O(\frac{k^2}{\gamma}\sqrt{n}m)$) but not in the preconditions (e.g., $n \geq 2D$).

LEMMA 5.2. *If $\gamma \geq 4$, then the Digraph Gap Algorithm (Basic Version) either verifies that $\kappa \geq k$ or finds a κ-separator.*

(i) *If $k \geq \sqrt{n}/2$ and $n \geq 2D$, the time is $O(\frac{D^2}{\gamma}\sqrt{n}m)$.*
(ii) *If $k \geq \sqrt{n}\log n$ and $n \geq 2D$, the time is $O(\frac{D^2\sqrt{n}m}{\gamma \log_{4n/\delta} n})$.*
(iii) *In general, the time is $O(\frac{n^{9/2}}{\gamma \log(n-\delta)})$.*
(iv) *If $\gamma \geq \epsilon D$ for some fixed $\epsilon > 0$, the time is $O((\sqrt{n} + D)\sqrt{n}m)$.*

PROOF. For each part we verify that the estimates for Steps (3) and (4) given above are within the claimed bound.

(i) The assumption $k \geq \sqrt{n}/2$ with $D \geq k$ shows $kn = k\sqrt{n}\sqrt{n} = O(k^2\sqrt{n}) = O(D^2\sqrt{n})$. Hence, the time for Step (3) is $O((kn)m/\gamma) = O(D^2\sqrt{n}m/\gamma)$ as claimed.

The assumption $n \geq 2D$ shows that in Step (4), $|R| = 2D$. Thus, $r = |R| = 2D \geq D + k$. Hence the number of max flow computations in Step (4) is $O(D^3/(\gamma D)) = O(D^2/\gamma)$. Each such computation uses time $O(\sqrt{n}m)$ giving (i).

(ii) The assumption on k gives $D \geq k \geq \sqrt{n}\log n$. Similar to part (i), $kn = k\sqrt{n}\sqrt{n} = O(k^2\sqrt{n}/\log n) = O(D^2\sqrt{n}/\log n)$. Hence $O((kn)m/\gamma) = O(D^2\sqrt{n}m/\ (\gamma \log n))$, which is within the desired time bound.

For Step (4), part (i) shows there are $O(D^2/\gamma)$ flow computations. Each flow is computed in time $O(\sqrt{n}m/\log_{4n/\delta} n)$ using Proposition 2.9(i). This gives the desired bound.

(iii) Step (3) uses time $O(knm/\gamma) = O(n^4/\gamma)$. This is within the desired bound since $\sqrt{n} \geq \log(n - \delta)$.

The time for Step (4) is estimated exactly as in Lemma 2.10(iii); for convenience we reproduce it. When $n < 2D$, Step (4) has $R = V$. Hence, $r = n$ and the number of flow computations is $O(n^3/(\gamma(n - k))) = O(n^3/(\gamma(n - \delta)))$ (since $\delta \geq k$). This bound also holds when $n \geq 2D$ since in the previous parts our bound was $O(D^2/\gamma)$ flows and by definition $D \le \delta \le n$. So the desired bound follows if we compute each flow using Proposition 2.9(ii), in time $O(n + (n-\delta)^{5/2}/\log(n-\delta))$.

(*iv*) By assumption $\gamma \ge \epsilon D \ge \epsilon k$. Hence, Step (3) uses time $O(knm/\gamma) = O(nm)$. Now it suffices to show Step (4) does $O(D)$ max flow computations.
The number of flow computations is, from (4), $O(r^3/\rho\rho')$. Step (4) always gives $r \le 2D$. Also $\rho' \ge \rho = \tau \ge \epsilon D/2$. So the number of flow computations is $O(D^3/(\epsilon D)^2) = O(D)$ as desired. $\square$

5.3. Gap Algorithm for Low Connectivities. This section presents the Flow Version of the Directed Gap Algorithm. The Flow Version is a modification of last section's Basic Gap Algorithm. It is more efficient when $k \le \sqrt{n}$. Just as in the Flow Version of the Gap Enlarger, it takes advantage of cheap max flows when k is small. The idea is to use Lemma 4.2(iv) to replace the rooted connectivity computations in Step (3) of the Basic Gap Algorithm by (conditional) max flow computations.

The hypothesis of Lemma 4.2(iv) requires a large value of the quantity $\kappa^+_W(S)$ (or symmetrically a large value of $\kappa^-_W(S)$). The following lemma shows how we can apply the Directed Nesting Lemma to handle the case when these values are not large (see the hypotheses on $\kappa^+_W(S)$, $\kappa^-_W(S)$).

Lemma 5.3. *Consider parameters k and τ satisfying*

$$\delta \ge k + 2\tau, \quad k > \kappa$$

and a set of vertices S satisfying

$$|S| \ge 2(k+\tau), \quad k+\tau \ge \kappa^+_W(S),\ \kappa^-_W(S).$$

Let $\kappa^+_W(S)$ ($\kappa^-_W(S)$) correspond to the separation triple (C^+, A^+, B^+) $((C^-, A^-, B^-))$ respectively. Either

(*i*) *some set $T \in \{S,\ A^+,\ B^+,\ C^+,\ A^-,\ B^-,\ C^-\}$ has $|T| > \kappa$ and $\kappa_W(T) = \kappa$, or*

(*ii*) *$R = (B^+ \cup C^+) \cup (A^- \cup C^-)$ is a τ-rich set of $\le 4(k+\tau)$ vertices.*

Proof. First observe that separation triples for $\kappa^+_W(S)$ and $\kappa^-_W(S)$ exist, since $|S| > k + \tau \ge \kappa^+_W(S), \kappa^-_W(S)$.

Assume possibility (i) of the lemma does not hold. We will derive three inequalities (displayed below) from the separation triple (C^+, A^+, B^+). We will invoke symmetry to get three similar inequalities from (C^-, A^-, B^-). These six inequalities will give the desired conclusion (ii).

Since $S \subseteq A^+ \cup C^+$, $|A^+| \ge |S| - |C^+| \ge 2(k+\tau) - (k+\tau) \ge k > \kappa$. Hence, if $A^+ \cap A^* = \emptyset$ for some κ-separation triple (C^*, A^*, B^*), then $\kappa^-_W(A^+) = \kappa$. Since we assume (i) fails for $T = A^+$, we get that $A^+ \cap A^* \ne \emptyset$ for every κ-separation triple (C^*, A^*, B^*).

Apply the Directed Nesting Lemma 5.1 to the separation triple (C^+, A^+, B^+) with B^+ in-extreme. Suppose Lemma 5.1(i) holds, that is, A^+ contains a κ^+-shore. This implies $\kappa^-_W(C^+) \le \kappa$. It is easy to see that the failure of (i) for $T = S$ shows $|C^+| > \kappa$. So we get that C^+ satisfies (i), a contradiction. We conclude that Lemma 5.1(ii) holds.

We claim that for every κ-separation triple (C^*, A^*, B^*),

$$|(B^+ \cup C^+) \cap A^*| \geq \tau.$$

In proof, Lemma 5.1(ii) shows the left-hand side is $\geq |B^+|$, so it suffices to show $|B^+| \geq \tau$. A vertex of B^+ has $\geq \delta \geq k+2\tau$ in-neighbors, all belonging to $B^+ \cup C^+$. Hence, $|B^+| + |C^+| \geq k + 2\tau$. This, together with the lemma's hypothesis

$$|C^+| = \kappa^+_W(S) \leq k + \tau,$$

implies the desired inequality.

Lemma 5.1(ii) also implies $B^+ \cap B^* = \emptyset$ for every κ-separation triple (C^*, A^*, B^*). Hence, if $|B^+| \geq k$, then $\kappa^+_W(B^+) = \kappa$ and (i) holds for $T = B^+$. Thus,

$$|B^+| < k.$$

A symmetric argument applies to the separation triple (C^-, A^-, B^-), which has A^- out-extreme. This gives analogs of the three inequalities displayed above, specifically

$$|(A^- \cup C^-) \cap B^*| \geq \tau, \quad |C^-| \leq k + \tau, \quad |A^-| < k.$$

For completeness, we state the analogous intermediate steps of the symmetric derivation:

$B^- \cap B^* \neq \emptyset$ for every κ-separation triple (C^*, A^*, B^*) (else $\kappa^+_W(B^-) = \kappa$).
If B^- contains a κ^--shore, then $\kappa^+_W(C^-) \leq \kappa$.
$|A^-| \geq \tau$.
$A^- \cap A^* = \emptyset$ for every κ-separation triple (C^*, A^*, B^*); $\kappa^-_W(A^-) = \kappa$.

Together the six inequalities show that R is a τ-rich set of $<4(k + \tau)$ vertices, that is, (ii) holds. □

We now give some preliminary remarks on the Flow Version of the Gap Algorithm. As in the Flow Version of the Gap Enlarger, we use conditional computations of $\kappa(x)$. We also use conditional computations of $\kappa_W(S)$. These are defined similarly: If the algorithm has found κ or $|S| \leq \kappa$ or $\kappa_W(S) > \kappa$, the computation can return $\min\{|S|, n-1\}$ or any valid separation triple for S and its corresponding value. On the other hand if κ has not been found and $|S| > \kappa$ and $\kappa_W(S) = \kappa$, then the computation must return the correct value $\kappa_W(S) = \kappa$ along with a corresponding κ-separator.

We use Lemma 4.2(i) to do conditional computations. Clearly, any algorithm that does some of its computations conditionally still maintains (C_0, A_0, B_0) as a valid separator. (As before, the algorithm maintains (C_0, A_0, B_0) as a triple having the smallest separator C_0 seen so far.) Furthermore, the algorithm actually finds the vertex connectivity κ, if at some point it conditionally computes a value $\kappa(x)$ or $\kappa_W(S)$ whose true value is κ.

As in the Basic Directed Gap Algorithm, the Flow Version assumes $\gamma \geq 4$, where the definitions of D, γ, τ are unchanged. We make one more assumption on the input to the Flow Version (for convenience): $n \geq 4D$.

As in the Basic Algorithm, the graph H is initially G, and gets edges added to it. Vertex sets X and Y are as in the Basic Algorithm. Invariants I1–I2 still hold. As in the Basic Algorithm, I1 (conformality) ensures that even when connectivity computations are done in H, the κ-separation triples and rich sets (and even superrich sets) are valid for G.

We use the following convention for specifying the graph in connectivity quantities $\kappa_W(\cdot)$, $\kappa^+_W(\cdot)$, etc.: When no graph is specified, for example, $\kappa_W(S)$, we're using the given graph G. Otherwise, the graph is explicitly specified, for example, $\kappa_W(C; H)$. Roughly speaking, Step (3) uses graph H, and the other steps use G. Actually, we only use the "no graph" notation in Steps (1)–(2), where the only graph in existence is G.

The Flow Version of the Gap Algorithm has the same outline as the previous Gap Algorithms: Steps (1)–(3) either find κ or find a τ-rich set R, which is processed in Step (4). The Flow Version is more involved than the Basic Version because (as already mentioned) the main tool, Lemma 4.2(iv), requires a large value $\kappa^+_W(S)$. Step (2) (Step (3)) handles the case where this large value is unavailable (available) respectively. Step (4) uses the Expander Lemma 2.8 on a τ-rich set of $\leq 4D$ vertices. Step (2) uses Lemma 5.3 to either find κ or find the desired τ-rich set. Step (3) uses Lemma 4.2(iv) and the Directed Nesting Lemma 5.1 to either find κ or make progress, as in the Basic Version, by adding edges that don't change the κ-separators (i.e., they maintain conformality).

Directed Gap Algorithm, Flow Version

Step (1). If $n = \delta + 1$, then the graph is k-connected so return. Otherwise, let S be an arbitrary set of $2D$ vertices. Initialize (C_0, A_0, B_0) to a separation triple that corresponds to a vertex of in- or out-degree δ, so $|C_0| = \delta$.

Find $\kappa_W(S)$. If a corresponding separation triple exists, update (C_0, A_0, B_0). If $\kappa^+_W(S)$ and $\kappa^-_W(S)$ are both $<k + \tau$ go to Step (2), else go to Step (3). /* As in the Basic Version, set S never changes. */

Explanation. As in previous versions the initial test handles complete graphs, so we can assume the graph is not complete and separators exist. Observe that

$$|S| = 2D > D > k + \tau. \tag{5}$$

Suppose $k > \kappa$. So (5) shows $|S| > \kappa$. Hence, if $\kappa_W(S) = \kappa$ the algorithm finds κ and a corresponding separator. In the opposite case, Lemma 4.1 shows S is 1-superrich.

We assume S is 1-superrich in the rest of the algorithm. This allows us to use S for conditional connectivity computations based on Lemma 4.2(i). In more detail, in the rest of the algorithm every conditional connectivity computation is carried out as follows. The computation will be for a quantity $\kappa_W(U)$. (In Step (2), the graph is G, in Step (3) it is H.) We compute $\kappa_W(U)$ conditionally using Lemma 4.2(i) and our 1-superrich set S, by computing

$$\min\{\kappa_W(U, y),\ \kappa_W(y, U) : y \in S\}. \tag{6}$$

Lemma 4.2(i) guarantees that, if $\kappa_W(U) = \kappa$ and $|U| > \kappa$, then the above minimum is κ.

Step (2). /* $\kappa^+_W(S), \kappa^-_W(S) < k + \tau$. */ Let $\kappa^+_W(S)$ ($\kappa^-_W(S)$) correspond to the separation triple (C^+, A^+, B^+) $((C^-, A^-, B^-))$ respectively. Conditionally compute

$\kappa_W(T)$ by following (6) for

$$T = A^+,\ B^+,\ C^+,\ A^-,\ B^-,\ C^-,$$

and update (C_0, A_0, B_0) for the separation triples that are found. Then, set $R = (B^+ \cup C^+) \cup (A^- \cup C^-)$ and go to Step (4).

Explanation. We will apply Lemma 5.3. Its hypotheses are satsfied because $\delta \geq D$, we assume $k > \kappa$, $|S|$ satisfies (5) and $\kappa_W^+(S)$, $\kappa_W^-(S)$ satisfy Step (2)'s hypothesis. If part (i) of the lemma holds, then one of the sets T of Step (2) has $|T| > \kappa$ and $\kappa_W(T) = \kappa$ (as in the lemma). So the conditional computation of $\kappa_W(T)$ is correct by Lemma 4.2(i), and the algorithm finds κ and a κ-separator. If part (ii) holds, then the algorithm goes to Step (4) with R a τ-rich set of $\leq 4(k+\tau) < 4D$ vertices.

Step (3). /$*$ One or both of $\kappa_W^+(S)$ and $\kappa_W^-(S)$ is $\geq k+\tau$. Assume $\kappa_W^-(S) \geq k+\tau$; if only $\kappa_W^+(S) \geq k+\tau$ then proceed in a symmetric manner, applying Steps (3.1)–(3.4) to the reverse graph. (If both $\kappa_W^+(S), \kappa_W^-(S) \geq k+\tau$ we could actually go directly to Step (4) as in the Basic Directed Gap Algorithm.) Recall that Step (3) is pledged to preserve invariants I1–I2. $*$/

Initialize the set T to S, vertex sets X and Y to $\emptyset$, and graph H to G. Repeat Steps (3.1)–(3.4) until they halt or go on to Step (4): /$*$ These steps enlarge both the graph H and the sets X and Y, similar to the Basic Version. $*$/

Step (3.1). If $T = \emptyset$, then set R to S and go to Step (4). Otherwise let b be a vertex of T. If $\kappa_W(S-b, b; H) \geq k+\tau$ then delete b from T and repeat Step (3.1). Otherwise, $(\kappa_W(S-b, b; H) < k+\tau)$ go to Step (3.2). /$*$ The graph H changes every time Step (3.1) is executed. The set T is changed only by the deletions of Step (3.1). A given vertex b can be chosen more than once in different executions of Step (3.1), as long as it is not deleted from T. $*$/

Explanation. We claim that if T becomes empty, the algorithm goes to Step (4) with $R = S$ a τ-rich set. In fact, we'll show a stronger property, superrichness, that is, every κ-separation triple (C^*, A^*, B^*) satisfies

$$|S \cap A^*| \geq \tau, \quad |S \cap B^*| \geq \tau.$$

As usual we assume $\kappa < k$.

For the first inequality, the argument is similar to Lemma 4.1: Suppose some κ-separation triple (C^*, A^*, B^*) has $|S \cap A^*| < \tau$. Since every vertex has out-degree at least δ, $|A^*| + |C^*| > \delta$. Hence $|A^*| > \delta - \kappa > D - k = \gamma > \tau$. Together with the supposition this implies $A^* - S \neq \emptyset$. So $(C^* \cup (A^* \cap S), A^* - S, B^*)$ is a separation triple. This makes $\kappa_W^-(S) \leq |C^* \cup (A^* \cap S)| < \kappa + \tau < k + \tau$. But this negates the assumption made in the comment for Step (3).

Next we show that the second inequality holds if T becomes empty. Suppose $|S \cap B^*| < \tau$ for some κ-separation triple (C^*, A^*, B^*). Initially T contains a vertex $b \in B^*$ since S is 1-superrich. Invariant I1 shows (C^*, A^*, B^*) is a separation triple of H. Lemma 4.2(v) applied to (C^*, A^*, B^*) shows shows $\kappa_W(S-b, b; H) \leq \kappa + |(S-b) \cap B^*| < k+\tau$. Hence, Step (3.1) never deletes b from T, and T never becomes empty.

Step (3.2). /$*$ $\kappa_W(S-b, b; H) < k+\tau$. $*$/ Apply Lemma 4.2(iv) to calculate $\kappa^-(b; H)$ as $\min\{\kappa(y, b; H) : y \in S\}$. Let (C, A, B) be a separation triple corresponding to $\kappa^-(b; H)$. Update (C_0, A_0, B_0) for this triple.

Explanation. For the calculation to be valid, we must check that the hypothesis of Lemma 4.2(iv) holds, that is, $\kappa^-(b; H) < \kappa^-_W(S; H)$. First, observe that $\kappa_W(S - b, b; H)$ corresponds to a separation triple, since from (5), $|S - b| = |S| - 1 \geq k + \tau > \kappa_W(S - b, b; H)$. This justifies the first inequality in the string

$$\kappa^-(b; H) \leq \kappa_W(S - b, b; H) < k + \tau \leq \kappa^-_W(S) \leq \kappa^-_W(S; H). \tag{7}$$

The last three inequalities are justified by the assumption of Step (3.2), the assumption of Step (3), and the fact that H is the result of adding edges to G, respectively. Equation (7) shows the desired hypothesis.

The separation triple (C, A, B) has A out-extreme. Hence, we can apply the Directed Nesting Lemma 5.1 to (C, A, B) and A. This is done in Steps (3.3)–(3.4).

Step (3.3). Conditionally compute $\kappa_W(C; H)$ by following (6). Update (C_0, A_0, B_0) accordingly.

Explanation. We can assume $|C| > \kappa$ since otherwise κ has been found. Hence, Lemma 4.2(i) shows the conditional computation of $\kappa_W(C; H)$ is valid.

If B contains a κ^--shore, then we have found κ (i.e., $\kappa^+_W(C; H) = \kappa$). So we can assume the opposite, that is, part (i) of Lemma 5.1 does not hold. Hence, for the next step (3.4), part (ii) of the lemma holds.

Step (3.4). If $|A| \geq k$ execute Step (3.4.1). Otherwise, execute Step (3.4.2), which is a symmetric step with A replaced by B. In both cases after executing (3.4.1) or (3.4.2), go back to Step (3.1) to start another iteration.

Explanation. The two cases are symmetric for two reasons. First, at least one of the sets A, B has size $\geq k$. This is implied by the inequality $n \geq |C| + 2k$, which holds because $n \geq 4D > 3D > (k + \tau) + 2k > \kappa^-(b; H) + 2k = |C| + 2k$. (The upper bound on $\kappa^-(b; H)$ was proved in (7).)

Second, the two steps are based on part (ii) of Lemma 5.1, which is symmetric in A and B (in fact they only use the first assertion of part (ii), that is, $A \cap A^*$ or $B \cap B^*$ is empty).

Step (3.4.1). Conditionally compute $\kappa_W(A; H)$ by following (6), and if appropriate update (C_0, A_0, B_0). Add B to X. If this makes $|X| > k$, then conditionally compute $\kappa_W(X; H)$ by following (6), update (C_0, A_0, B_0) if appropriate and halt. Otherwise, add edge (x, y) to H for every $x \in V - y$, $y \in B$ where this edge does not exist.

Explanation. The assumption $|A| \geq k > \kappa$ shows the conditional computation gives the true value of $\kappa_W(A; H)$ if this true value is κ. So we can now assume $\kappa^-_W(A; H) > \kappa$. Thus, every κ-separation triple (C^*, A^*, B^*) has $A \cap A^* \neq \emptyset$. Lemma 5.1(ii) implies $B \cap B^* = \emptyset$. Hence, the additions to X and H preserve Invariants I1–I2. Also when $|X| > k$ the algorithm halts having found κ, by I1.

Step (3.4.2). This is symmetric to Step (3.4.1), interchanging A and B. (We add A to Y and add edge (x, y) for every $x \in A$, $y \in V - x$.)

Explanation. As in the Basic Version, it is not obvious why Step (3) cannot loop indefinitely. The explanation is similar to the Basic Version and is given below in (a′).

Step (4). /* If κ has not been found and $\kappa < k$, then R is now τ-rich with at most $4D$ vertices. */ Add vertices to make $|R| = 4D$. Apply the Expander Lemma 2.8 with $\rho = \tau$, constructing the graph $\chi(R, d)$ and computing the values $\kappa(x, y; G)$ specified in the lemma (for digraphs). Update (C_0, A_0, B_0) for these values.

Explanation. If Step (4) starts with $\kappa < k$ and κ still not found, the Expander Lemma guarantees that Step (4) computes κ.

As with the Basic Version, we have verified this algorithm is correct as long as it halts, that is, Step (3) does not loop indefinitely. We continue to follow the previous derivation by giving analogs of facts (a)–(b) of the Basic Version:

(a′) the loop of Step (3) executes Steps (3.1)–(3.4) $O(k/\tau)$ times;
(b′) the graph H always has $m(H) = O(m)$.

(a′) shows the algorithm halts.

PROOF OF (a′). The argument is essentially that of the Basic Version. We go through it quickly, any missing details are the same as the proof of (a) (before Lemma 5.2).

It suffices to show that each execution of Step (3.4) enlarges X or Y by at least τ vertices. Step (3.2) sets (C, A, B) to a separation triple corresponding to $\kappa^-(b; H)$. We analyze the case that Step (3.4.1) adds B to X. Step (3.4.2) is symmetric, adding A to Y, so the same argument applies.

Invariant I2 shows a vertex of X has in-degree $n - 1$. Hence, $X \cap B = \emptyset$. So we need only show $|B| \geq \tau$. Recall $|C| = \kappa^-(b; H) < k + \tau$ (as shown in (7)). This plus the fact that every degree is $\geq \delta \geq D = k + \gamma$ gives

$$|B| \geq \delta + 1 - |C| > (k + \gamma) - (k + \tau) = \tau. \quad \square$$

PROOF OF (b′). This was proved in the first paragraph of the proof of (b): Step (3.4.1) only adds edges when $|X| \leq k$ for the newly enlarged set X, and similarly for Step (3.4.2). Hence, $O(kn) = O(\delta n) = O(m)$ edges are added to H. $\square$

LEMMA 5.4. *If $\gamma \geq 4$ and $n \geq 4D$, then the Flow Version of the Directed Gap Algorithm either verifies that $\kappa \geq k$ or finds a κ-separator. The time is $O((n + k^3/\gamma)m)$.*

PROOF. We have verified the Flow Version is correct so we need only estimate the time. We allocate $O(m)$ time for bookkeeping. We will see that in each step the remaining time is dominated by computations of connectivity quantities $\kappa(x)$, $\kappa_W(S)$, $\kappa(x, y)$. Hence, we only estimate the time for computing these quantities.

Step (1) uses rooted connectivity computations to find $\kappa_W^+(S)$ and $\kappa_W^-(S)$ in time $O(nm)$.

Steps (2)–(4) compute all connectivity quantities using max flow computations, because of conditional computations. We claim each of these steps performs a total of

$$O(D^2/\gamma)$$

max flow computations, and furthermore each computation can stop if the flow value reaches D. This claim implies that the time for a max flow computation is $O(Dm)$. (Note that the flow graph always has $O(m)$ edges, since it is either G or H, and $m(H) = O(m)$ by property (b′).) Since $D \leq 2k$, the claim implies the flow computations take total time $O(D^3 m/\gamma) = O(k^3 m/\gamma)$. This gives the desired time bound for the entire algorithm.

Step (2) computes $O(1)$ quantities $\kappa_W(T)$ conditionally. This gives $O(|S|) = O(D)$ flow computations. This bound is satisfactory, since by definition $D \geq \gamma$ which implies $O(D) = O(D^2/\gamma)$. Furthermore, each quantity $\kappa_W(T)$ is only used to update (C_0, A_0, B_0). So the quantity is irrelevant if it is larger than κ. Hence, each flow computation can stop if the flow value reaches D, since $D \geq k > \kappa$. The claim for Step (2) follows.

Consider a flow computation in Step (3.1). By definition it can be stopped when the flow value reaches $k + \tau < D$. If the value reaches $k + \tau$, then Step (3.1) deletes b from T. This occurs at most $|S| = 2D$ times. So these computations contribute $O(D)$ flows, which as just shown is within the desired bound.

Every other flow computation of Step (3.1) is followed by an execution of Steps (3.2)–(3.4). This constitutes one repetition of the loop of Step (3). We will show that one execution of Steps (3.2)–(3.4) performs $O(D)$ flow computations, each of which can stop at flow value D. Property (a′) shows there are $O(k/\tau) = O(D/\gamma)$ such executions. This gives $O(D^2/\gamma)$ flow computations total. The claim for Step (3) follows.

Step (3.2) computes $\kappa^-(b; H)$ using $|S| = 2D$ flow computations. Equation (7) shows each computation can be stopped if the flow reaches value $k + \tau \leq D$.

Steps (3.3) and (3.4) are similar to Step (2): They compute $O(1)$ quantities $\kappa_W(\cdot)$ conditionally, and the quantities are only used to update (C_0, A_0, B_0). As shown for Step (2) this gives $O(|S|) = O(D)$ flow computations total, and each computation can stop if the flow value reaches D. This completes the analysis of Step (3).

Finally consider Step (4). Exactly as in the Basic Algorithm this step does $O(r^3/\gamma(r - k))$ flow computations. (In detail, Step (4) performs $2m(\chi(R, d))$ max flow computations of quantities $\kappa(x, y)$. Let $r = |R|$. Equation (4) with $\rho = \tau = \Theta(\gamma)$ gives the stated bound on flow computations.) Since $r = |R| = 4D > D + k$ the stated bound is $O(D^3/(\gamma D)) = O(D^2/\gamma)$. The flow computations are used only to update (C_0, A_0, B_0) so they can be stopped at value D. The claim for Step (4) follows. □

5.4. CHECKING ***aδ***-CONNECTEDNESS. We complete Section 5 by showing that for digraphs and any fixed constant $a < 1$, $a\delta$-connectedness can be checked in time $O((\kappa + \sqrt{n})\sqrt{n}m)$. We start with an algorithm that achieves the desired time bound for the case $a = 1/2$:

$\delta/2$-connectedness Checker

Step (1). If $\delta < 16$ then find κ using the algorithm of Henzinger et al. [2000] and return.

Step (2). Initialize k to $k_0 = \min\{\delta/2, \sqrt{n}\}$. Then repeatedly execute Steps (2.1)–(2.2) until Step (2.1) returns.

Step (2.1). Execute the Basic Directed Gap Algorithm to check k-connectedness. Return if the algorithm finds κ or if $k = \delta/2$, that is, the algorithm has verified that $\kappa \geq \delta/2$.

Step (2.2). Set $k = \min\{2k, \delta/2\}$.

We must show the Gap Algorithm works correctly, and we must also verify the desired time bound. We do these together. First note that Step (1) uses (less than) $O(\kappa nm)$ time in general [Henzinger et al. 2000], which in Step (1) becomes $O(nm)$. This bound is satisfactory.

For Step (2), we first show that each execution of the Gap Algorithm works correctly and uses time $O((\sqrt{n}+k)\sqrt{n}m)$. Step (2.1) always has $k \le \delta/2$. This implies that $D = 2k$ and $\gamma = k$ in the Gap Algorithm. The Gap Algorithm also has $\gamma = k \ge k_0 \ge \min\{16/2, \sqrt{16}\} = 4$, as well as $\gamma \ge D/2$. So all hypotheses of Lemma 5.2(iv) are satisfied. Now the lemma implies the above properties of the Gap Algorithm.

The first execution of Step (2.1) has $k = k_0 \le \sqrt{n}$, so its time is satisfactory. Suppose there is more than one execution. Clearly, this means $k_0 = \sqrt{n}$. Hence, every iteration uses time $O(k\sqrt{n}m)$.

Each execution after the first has $k \le 2\kappa$, by the verification performed in the previous execution. Summing the time $O(k\sqrt{n}m)$ over all values of k (with k doubling and $\le 2\kappa$) gives total time $O(\kappa\sqrt{n}m)$. (The fact that k may not have doubled for the last execution just makes our time bound an overestimate.)

We can now give the general algorithm to check $a\delta$-connectedness for any fixed $a, 0 < a < 1$.

$a\delta$-connectedness Checker

Step (1). Check $\delta/2$-connectedness using the previous algorithm. Return if it finds κ or if $a \le 1/2$.

Step (2). If $(1-a)\delta \le 4$ then find κ using the algorithm of Henzinger et al. [2000]. Then return.

Step (3). Check k-connectedness using the Basic Directed Gap Algorithm with $k = a\delta$. Then return.

As before, we show the Gap Algorithm works correctly and we verify the desired time bound. The time bound for Steps (1)–(2) is trivial. (Step (1) has been done. In Step (2), $\kappa \le \delta = O(1)$, so again the time bound of Henzinger et al. [2000] is $O(\kappa nm) = O(nm)$.) These steps show that Step (3) has $a > 1/2$ and $(1-a)\delta > 4$. The former makes $k > \delta/2$, so the Gap Algorithm has $D = \delta$ and $\gamma = D - k = (1-a)\delta > 4$. Also $\gamma = (1-a)\delta = (1-a)D$, a positive constant times D. Thus, all hypotheses of Lemma 5.2(iv) are satisfied, and the algorithm works correctly. Since Step (1) shows $\kappa \ge \delta/2 = D/2$, the time bound of Lemma 5.2(iv) is satisfactory.

THEOREM 5.5. *For any fixed $a < 1$, we can check $a\delta$-connectedness in time $O((\kappa+\sqrt{n})\sqrt{n}m)$. In an undirected graph, the same bound holds with m replaced by κn. The space is $O(m)$.*

PROOF. The $a\delta$-connectedness Checker achieves the bound of the theorem for digraphs. For undirected graphs we make two changes to the $a\delta$-connectedness Checker:

First, Step (1) uses Henzinger's algorithm to check $\delta/2$-connectedness [Henzinger 1997]. It runs in time $O(\min\{\kappa, \sqrt{n}\}n^2)$, and even $O(\kappa n^2)$ is within the theorem's bound (the latter includes the term $\sqrt{n}\sqrt{n}\kappa n$).

Second, before Step (2), we reduce the graph to $O(\delta n)$ edges, as follows: find a maximal forest decomposition of the given graph. Discard all edges except those in the first $\delta + 1$ forests. The new graph has the same minimum separators and it has $O(\delta n)$ edges. The algorithm of Nagamochi and Ibaraki [1992] accomplishes this in time $O(m)$. Since $\kappa \ge \delta/2$ in Steps (2)–(3), the term m in the digraph time bound can now be replaced by κn. (If desired the undirected Gap Algorithm can be used in Step (3), since Lemma 2.10(iv) shows it satisfies the time bound of Lemma 5.2(iv).) □

When $\kappa > n^{2/5}$ the theorem's bound is superior to our general bound to find κ. (In the opposite case the two bounds are the same, $O(nm)$.)

For completeness, we give an alternate algorithm for checking $\delta/2$-connectedness in undirected graphs. The time bound is $O(\kappa n^2)$. (So this bound also suffices for Theorem 5.5.) Actually, for any $k \le \delta/2$, we show how to check k-connectedness in time $O(kn^2)$. Using this procedure with k doubling gives our algorithm for checking $\delta/2$-connectedness.

Undirected k-connectedness Checker, $k \le \delta/2$

Apply the algorithm of Nagamochi and Ibaraki [1992] to get a maximal forest decomposition of the given graph. Let $S \subseteq V$ consist of the last $\delta - k$ vertices to be labelled. Let H be the subgraph consisting of the first k forests. Compute $\kappa_W(S; H)$. If this quantity is strictly less than k, then it equals κ. Otherwise, $\kappa \ge k$.

To show this algorithm is correct, suppose $\kappa < k$. This implies $\kappa_W(S; G) = \kappa$, for two reasons: (i) $|S| = \delta - k \ge k$. (ii) Any two vertices $x, y \in S$ have $\kappa(x, y) \ge k$ [Henzinger 1997]. Hence, any κ-separation triple (C^*, A^*, B^*) has S contained in C^* plus one of the shores.

Finally, note that by definition G and H have the same separators of $<k$ vertices [Nagamochi and Ibaraki 1992]. Hence, $\kappa_W(S; H) = \kappa_W(S; G) = \kappa$.

The time is $O(kn^2)$, since Henzinger et al. [2000] computes $\kappa_W(S; H)$ in time $O(nm(H)) = O(kn^2)$.

6. *Wrap-Up*

This section completes the algorithm to find the connectivity of a graph, and derives the time bound $O((n + \min\{\kappa^{5/2}, \kappa n^{3/4}\})m)$. It may be helpful to view this as the equivalent bound, $O((n+\kappa \min\{\kappa^{3/2}, n^{3/4}\})m)$. For undirected graphs, we'll replace m by κn in these bounds.

The previous sections have developed our tools, the Gap Enlarger and the Gap Algorithm. The main issue now is choosing a good gap parameter Δ for the Enlarger. Section 6.1 presents the overall connectivity algorithm. Section 6.2 derives the time bound for the algorithm.

6.1. THE COMPLETE CONNECTIVITY ALGORITHM. We state the main algorithm to find the connectivity κ. Let G be an arbitrary given graph, directed or undirected. The algorithm returns $\kappa(G)$, plus a corresponding separation triple if G is not complete.

As we present the algorithm, we also verify the correctness and time bound of all steps except for the algorithm's core, Step (3). A partial analysis of Step (3) is given, and the analysis is completed in the next section.

Main Algorithm

Step (0). If G is undirected, then do the following preprocessing: Find a maximal forest decomposition and discard all edges except those in the first $\delta + 1$ forests. /* We continue to call this reduced graph G. */

Explanation. The reduced graph has the same minimum separators as the original and has $O(\delta n)$ edges. This leads to the reduced time bound for undirected graphs.

Using the algorithm of Nagamochi and Ibaraki [1992] the time for preprocessing is $O(m)$. This is within the desired bound for undirected graphs since $m = O(n^2)$.

Step (1). Apply Theorem 5.5 to check $\delta/2$-connectedness, that is, either find κ or verify that $\kappa \geq \delta/2$. Return if κ is found.

Explanation. Theorem 5.5 gives time $O((\kappa + \sqrt{n})\sqrt{n}m)$, with the m term appropriately reduced if G is undirected. If $\kappa \leq \sqrt{n}$, the time is $O(nm)$, which is satisfactory. If $\kappa > \sqrt{n}$, we have $\kappa^{3/2} > n^{3/4}$, so $\kappa\sqrt{n} \leq \kappa \min\{\kappa^{3/2}, n^{3/4}\}$. This shows the time is again satisfactory.

Step (2). If $\delta < 16$ or $\delta \geq n - n^{7/8}$, then compute κ using the algorithm of Henzinger et al. [2000] and return.

Explanation. In the first case, the time is $O(\kappa nm) = O(nm)$ for digraphs. Thus, the time is satisfactory for both directed and undirected graphs.

For the second case recall (from the remark after Lemma 3.3) [Henzinger et al. 2000] uses time $O((n(n-\delta))^2)$. Using Step (2)'s bound on δ this becomes $O(n^{15/4})$. Since Step (2) has $\kappa \geq \delta/2 = \Theta(n)$ and $m = \Omega(\delta n) = \Omega(n^2)$, the desired bound is also $O(n^{15/4})$.

The rest of the algorithm has

$$\kappa \geq \delta/2 \quad \text{and} \quad 16 \leq \delta < n - n^{7/8}. \tag{8}$$

The last inequality implies $\delta < n - 1$ so G is not complete and a κ-separation triple exists.

Step (3). Define a parameter Δ for gap enlargement by first setting

$$\Delta_0 = \begin{cases} \sqrt{\delta} & \delta \leq \sqrt{n} \\ \delta/n^{1/4} & \sqrt{n} < \delta \leq n^{2/3} \\ \frac{\delta}{n^{1/4}\log_{n/\delta} n} & n^{2/3} < \delta \leq n/3 \\ \frac{n^{3/4}}{\log n} & n/3 < \delta < n - n^{7/8} \end{cases}$$

and then defining $\Delta = \max\{4, \lfloor \Delta_0 \rfloor\}$.

/* The last case for Δ_0 may be empty, this causes no problem. Assuming the model of computation is a RAM machine, the last case calculates $\log n$ as $\lfloor \log n \rfloor$, and the third case calculates $\log_{n/\delta} n$ as $\lfloor \log n \rfloor / \lfloor \log (n/\delta) \rfloor$. We call attention to these floors when they arise in the analysis (they cause no problem). It is easy to see that none of the other arithmetic in the entire algorithm causes any problem on a RAM. */

Execute the Gap Enlarger with this value of Δ to either find κ or get the graph H with gap $\delta(H) - \kappa(H) = \delta - \kappa(H) \geq \Delta$. (If $\delta \leq \sqrt{n}$ use the Flow Version of the Gap Enlarger. If G is undirected (directed) use the algorithm of Section 3 (Section 4).)

Execute the Gap Algorithm on H with

$$k = \delta - \Delta.$$

TABLE II. PRECONDITIONS AND TIME BOUNDS FOR SUBROUTINES OF THE MAIN ALGORITHM

Row	Algorithm	Hypotheses	Time Beyond $O(nm)$
1	Gap Enlarger	$n > \delta + \Delta, \quad \beta = \dfrac{n-\Delta}{\delta}$	$O(\Delta n \min\{m,\ (n-\delta)^2\} \log_\beta n)$
2	Gap Enlarger, F.V.	$n \geq 2\Delta, \quad \sqrt{n} \geq \delta$	$O(\delta^2 \Delta m)$
3	Gap Alg. (F.V.)	$n \geq 4\delta + \Delta$	$O\left(\dfrac{\delta^3 m}{\Delta}\right)$
4	Gap Alg. (B.V.)	$n \geq 2\delta + \Delta, \quad k \geq \sqrt{n}/2$	$O\left(\dfrac{\delta^2 \sqrt{n} m}{\Delta}\right)$
5	Gap Alg. (B.V.)	$n \geq 2\delta + \Delta, \quad k \geq \sqrt{n} \log n$	$O\left(\dfrac{\delta^2 \sqrt{n} m}{\Delta \log_{4n/\delta} n}\right)$
6	Gap Alg. (B.V.)		$O\left(\dfrac{n^{9/2}}{\Delta \log (n(H)-\delta)}\right)$

B.V. = Basic Version, F.V. = Flow Version. We assume $D = \delta = \delta(H)$, $\gamma = \Delta$, $k = \delta - \Delta$, $n(H) = n - \Delta$, $m(H) \leq m$.

(If the graph is undirected use the algorithm of Section 2. If the graph is directed use an algorithm from Section 5—the Flow Version of the Directed Gap Algorithm if $\delta \leq \sqrt{n}$ and the Basic Version otherwise. Notice the use of n here and not $n(H)$, even though the graph is H.)

Compare two separation triples of G: the triple (C_0, A_0, B_0) found by the Gap Enlarger and the expansion of the triple for $\kappa(H)$ found by the Gap Algorithm. (The latter need not exist if $\kappa(H) \geq k$.) The smaller separator is the desired κ-separator for G.

Explanation. The next section shows that each execution of the Gap Enlarger and the Gap Algorithm works correctly and within the desired time bound. Assume for the moment that each execution works correctly. Then, we can show that Step (3) is guaranteed to find a κ-separator of G, as follows: Suppose the Gap Enlarger does not find κ. The Gap Enlarger initializes (C_0, A_0, B_0) to a separator of δ or fewer vertices, so we must have $\kappa(G) < \delta$. Thus, $\kappa(H) = \kappa(G) - \Delta < \delta - \Delta = k$ (Lemma 3.3(i)–(ii)). Hence, the Gap Algorithm, which checks k-connectedness, finds $\kappa(H)$ (Lemmas 2.10, 5.2). The expansion of the separator returned by the Gap Algorithm is the desired κ-separator of G (since again, $\kappa(G) = \kappa(H) + \Delta$).

Remark. In the definition of Δ_0, the exponent $2/3$ can be replaced by any exponent strictly between $1/2$ and 1. For undirected graphs, the second case ($\Delta_0 = \delta/n^{1/4}$) can be eliminated and incorporated into the third case.

6.2. ANALYSIS. This section analyzes the core of the connectivity algorithm, Step (3) of the Main Algorithm. It begins by summarizing all the Gap Enlarger and Gap Algorithms in Table II. Then, it analyzes Step (3) with the help of Table II. This analysis is in Lemmas 6.1–6.2. These lemmas examine each of the four ranges for δ in the definition of Δ_0, and show that in each case the algorithm works correctly and achieves the desired time bound.

6.2.1. *Derivation of Table II.* Table II unites and simplifies the preconditions and time bounds of the various Gap Enlargers and Gap Algorithms. The table expresses all the previous results in terms of the parameters of the Main Algorithm.

The table reduces the amount of cross-referencing and parameter-translation that would otherwise be needed to prove the lemmas of the next section.

We work through the table piece by piece. We first verify the parameter values that are specified in the caption of Table II. Then we describe the entries of the table, and verify that each entry follows from previous lemmas.

In the caption the parameters involving H come from Lemmas 3.3(i), 3.4(i) and Corollaries 4.3 and 4.4. The value for k is from Step (3) of the Main Algorithm. It remains to show $D = \delta$ and $\gamma = \Delta$.

The undirected Gap Algorithm defines $\gamma = \delta - k$, which with Step (3)'s setting of k becomes

$$\gamma = \Delta$$

as desired. Here, we're using the fact that $\delta(H) = \delta$, since the Gap Algorithm is executed on graph H. We now show the above equation holds in the directed case. The Directed Gap Algorithm defines $\gamma = D - k$ where $D = \min\{2k, \delta\}$. So it suffices to show

$$D = \delta,$$

which is the other remaining equation for Table II's caption. Since Step (3) chooses $k = \delta - \Delta$ for the Gap Algorithm the definition of D shows we want this inequality: $2(\delta - \Delta) \geq \delta$, that is,

$$\delta \geq 2\Delta. \tag{9}$$

We show (9) holds for both directed and undirected graphs.

The definition of Δ in Step (3) shows either $\Delta = 4$ or $\Delta \leq \Delta_0$. In the first case, we want $\delta \geq 2 \cdot 4$. This is implied by (8). For the second case, it suffices to check

$$\delta \geq 2\Delta_0$$

in each of the four ranges of the definition of Δ_0. For the lowest range, we want $\delta \geq 2\sqrt{\delta}$, i.e., $\delta \geq 4$, which holds by (8). For the remaining ranges, note (8) gives

$$n \geq 16.$$

The two middle ranges have $\Delta_0 \leq \delta/n^{1/4}$. Thus $\delta \geq n^{1/4}\Delta_0 \geq 2\Delta_0$. The highest range assumes $\delta > n/3$, so it suffices to have $n/3 \geq 2\Delta_0 = 2n^{3/4}/\log n$. This amounts to $n^{1/4}\log n \geq 6$, which again holds ($\log 16 = 4$). This last inequality holds even if we replace $\log n$ by its floor, so our computation on a RAM is valid. (9) now follows.

This completes the derivation of Table II's caption. Furthermore recall that the Gap Algorithm always requires $\gamma \geq 4$ (Lemmas 2.10, 5.2, 5.4). This assumption always holds since $\gamma = \Delta \geq 4$.

We turn to the body of Table II. It has an entry for each Gap Enlarger and Gap Algorithm. There is only one Gap Algorithm for undirected graphs, and it gives the entries in rows 3–6. For directed graphs, the Basic Version of the Gap Algorithm gives rows 4–6 and the Flow Version gives row 3.

The Hypotheses column of the table gives the inequalities that correspond to the preconditions of the algorithm. The Time column gives time bound for the algorithm. The bound is stated in terms of the parameters n, m, etc. of G (even for the Gap Algorithm, which is executed on H). To simplify the table we omit the term $O(nm)$ from many of the time bounds, as indicated by the column heading. This is fine for our purposes since the desired bound for the complete algorithm includes this term $O(nm)$ (recall the beginning of this section or see Theorem 6.3). Note also that for undirected graphs we now have $m = \Theta(\kappa n)$ because of the preprocessing Step (0) and Equation (8). So we no longer worry about the difference between the directed and undirected bounds in Theorem 6.3.

Now we verify that the table entries are correct. Rows 1–2 for the Gap Enlarger follow from Lemmas 3.3 and 3.4 respectively for undirected graphs. (Directed graphs are the same as undirected graphs, by Corollaries 4.3 and 4.4.) The hypotheses of the two lemmas that are omitted from the table ($\delta, \Delta > 0$ and $\delta > 2$ respectively) follow easily from (8). Finally note that the expression $n + (n - \delta)^2$ in the time bound of Lemma 3.3 has been simplfied by dropping the term n. This simplification is fine because it omits a term that contributes $O(nm)$ to the overall time bound. To see this, note the omitted term contributes Δn^2 to the overall time bound. Recall that $\Delta \leq \delta$ (by Lemma 3.3(i)–(ii)). Hence $\Delta n^2 \leq \delta n^2 \leq nm$.

The remaining rows are for the Gap Algorithm. It is always executed on H. The hypotheses come from the directed case – the undirected bound always has less restrictive assumptions. Note that we have already verified the hypothesis $\gamma \geq 4$. Row 3 comes from Lemmas 5.4 and 2.10(i), using parameter values in the caption (in particular we use $n(H) = n - \Delta$ rather than n). Similarly rows 4–6 come from Lemmas 5.2 and 2.10 parts (i)–(iii), respectively. In the hypothesis, for k in rows 4–5, for simplicity we replaced $n(H)$ by n—there's no harm in strengthening the hypothesis. In row 5, notice the time bound is increasing with n (since $\sqrt{n}/\log n$ is an increasing function for n sufficiently large), justifying our replacement of $n(H)$ by n. In row 6 we replaced $n(H)$ by n in the numerator but not the denominator.

Finally, note that in the time bounds of Table II Δ can be replaced by Δ_0. This only requires $\Delta = \Theta(\Delta_0)$, which is obvious if $\Delta_0 \geq 4$ and follows from $\Delta_0 = \Omega(1)$ otherwise. Furthermore, it is a simple matter to check that in the time bounds the floors of logs used to calculate Δ_0 in the two highest ranges can be ignored.

6.2.2. *Correctness and Efficiency of the Algorithm.* The next two lemmas prove that in all cases the algorithm works correctly and achieves the desired time bound. Each argument consists of checking the hypotheses from the rows of Table II that correspond to the relevant Gap Enlarger and Gap Algorithm, and verifying that the time specified by Table II is as desired. Note that once we have checked the hypotheses we know that Step (3) works correctly (for the case at hand). So the arguments omit mentioning that the algorithm is correct. Also the term $O(nm)$ which is omitted in Table II is included in the time bounds of the lemmas.

The first lemma treats the range of the smallest values for δ. This range corresponds to when Step (3) uses Flow Versions—the Flow Version of the Gap Enlarger, and the Flow Version of the Directed Gap Algorithm. The second lemma treats the three remaining ranges for δ. In these cases, Step (3) does not use the Flow Versions.

LEMMA 6.1. *If $\delta \leq \sqrt{n}$ then Step* (3) *finds κ and a κ-separator in time $O((n + \kappa^{5/2})m)$.*

PROOF. Step (3) executes the Gap Enlarger, Flow Version. We check the hypotheses of Table II row 2. We have $n \geq 2\Delta$ since $n \geq \delta \geq 2\Delta$ by (9). The second hypothesis $\sqrt{n} \geq \delta$ has been assumed. Hence, the Gap Enlarger uses time

$$O(\delta^2 \Delta m).$$

Next consider the Gap Algorithm. Step (3) executes the Flow Version if the graph is directed. We use row 3 of Table II, for both directed and undirected graphs. The hypothesis of row 3 holds since $n - \Delta \geq \delta^2 - \delta/2 \geq 15\delta > 4\delta$ (we've used the assumption $n \geq \sqrt{\delta}$ and (9), (8)). Row 3 gives the time as

$$O(\delta^3 m/\Delta).$$

Finally note that the two displayed quantities are within the lemma's time bound because $\kappa \geq \delta/2$ from (8) and $\Delta = \Theta(\sqrt{\delta})$. $\square$

LEMMA 6.2. *If $\delta > \sqrt{n}$, then Step* (3) *finds κ and a κ-separator in time $O(\kappa n^{3/4} m)$.*

Remark. The range $\delta > \sqrt{n}$ covered by the lemma encompasses several different cases of the algorithm, leading to the different cases in the lemma's proof. The cases result from the fact that several quantities tied to the algorithm's time bound vary radically as δ gets close to n. We've already mentioned the first such quantity, the time for computing $\kappa(x, y)$ (recall Table I). Here are two more:

(i) The factor $\log_\beta n$ in the time for gap enlargement (Lemma 3.3(iii)). We can approximate β as n/δ (see the argument below). When $\delta = \Theta(n^{1-\epsilon})$ we have $\beta = \Theta(n^\epsilon)$ and $\log_\beta n = \Theta(1)$. But $\log_\beta n$ grows as δ gets very close to n, for example, when $\delta = n - n^{7/8}$, we have $\beta \approx 1 + 1/n^{1/8}$ and $\log_\beta n \approx n^{1/8} \log n$.

(ii) The degree of the expander graph. Lemma 2.10(i) shows the degree is $O(\delta/\gamma)$ when $n \geq 2\delta$. When $n = 2\delta$ this becomes $O(n/\gamma)$. For larger values of δ, Lemma 2.10(iii) shows the degree is $O(n^2/(\gamma(n - \delta)))$. So when $\delta = n - n^{7/8}$ the degree is a factor $n^{1/8}$ larger than when $\delta = n/2$. Note these degrees are calculated independently of any time bounds for network flow, so this effect is different from the first effect that we mentioned.

PROOF. We consider the three highest ranges in the definition of Δ_0. Observe that all three ranges have

$$\Delta \leq n^{3/4} \leq n/2. \tag{10}$$

In proof, the first inequality follows from inspecting the definition of Δ_0, and also noting that $n \geq 16 = 4^2$ suffices when $\Delta = 4$. The second inequality also follows from $n \geq 16$.

Case (1). $\sqrt{n} < \delta \leq n^{2/3}$.

Consider the Gap Enlarger. We must check the first inequality of Table II row 1: $n \geq \delta^{3/2} \geq 4\delta > \delta + \Delta$ (using (8) and (9)). So the bound of row 1 applies. Using (10), $\beta = (n - \Delta)/\delta \geq n/(2\delta) \geq n/(2n^{2/3}) = n^{1/3}/2$. Hence row 1 gives time

$$O(\Delta n m \log_\beta n) = O\left(\frac{\delta}{n^{1/4}} nm\right) = O(\kappa n^{3/4} m).$$

We estimate the time for the Gap Algorithm using Table II row 4. We check the two hypotheses: From (10) and (8), $n - \Delta \geq n/2 \geq \delta^{3/2}/2 \geq 2\delta$. Also $k = \delta - \Delta \geq \delta/2 > \sqrt{n}/2$ (using (9)). Hence, row 4 gives the time as

$$O\left(\frac{\delta^2}{\Delta}\sqrt{n}m\right) = O\left(\frac{\delta^2}{\delta/n^{1/4}}\sqrt{n}m\right) = O(\kappa n^{3/4} m).$$

Case (2). $n^{2/3} < \delta \leq n/3$.

We use several times the fact that for any constant $c > 1/3$, $\log(cn/\delta) = \Theta(\log(n/\delta))$. This holds from the above assumption $\delta \leq n/3$. (This is obvious for $c \geq 1$. For $c < 1$, observe that we have $\log(cn/\delta) \geq \epsilon$ for some $\epsilon > 0$. This implies $\log(cn/\delta) \geq \epsilon' \log(n/\delta)$ for some $\epsilon' > 0$.)

Consider the Gap Enlarger. The hypothesis of Table II row 1 holds since

$$n \geq 3\delta > 2\delta + \Delta$$

by (9). To compute the time note that (10) gives $\beta = (n - \Delta)/\delta \geq n/(2\delta)$. So row 1 shows the time is

$$O(\Delta nm \log_\beta n) = O\left(\frac{\delta}{n^{1/4}\log_{n/\delta} n} nm \log_{n/2\delta} n\right) = O(\kappa n^{3/4} m).$$

For the Gap Algorithm, we use Table II row 5. Its first hypothesis is the inequality for n displayed above. For the second hypothesis, (9) implies $k = \delta - \Delta \geq \delta/2 > n^{2/3}/2 \geq \sqrt{n}\log n$ for sufficiently large n. (We can assume large n since we only need row 5 for the time bound. Correctness of the Gap Algorithm is already guaranteed by our verification that $\gamma \geq 4$. This can be seen from the algorithms, or just the fact that the hypothesis of row 6 is empty.) Hence, row 5 gives time

$$O\left(\frac{\delta^2\sqrt{n}m}{\Delta \log_{4n/\delta} n}\right) = O\left(\frac{\kappa^2\sqrt{n}m}{\frac{\kappa}{n^{1/4}\log_{n/\delta} n}\log_{4n/\delta} n}\right) = O(\kappa n^{3/4} m).$$

Case (3). $n/3 < \delta < n - n^{7/8}$.

The upper bound of this case follows from (8). The desired time bound is now $O(n^{15/4})$, since $\kappa \geq \delta/2 \geq n/6$ and $m \geq n\delta/2 \geq n^2/6$.

Consider the Gap Enlarger, Table II row 1. The hypothesis is satisfied since by (10), $n > \delta + n^{7/8} \geq \delta + n^{3/4} \geq \delta + \Delta$. Define ϵ so that

$$\delta = (1 - \epsilon)n.$$

The assumption of Case (3) implies $1/n^{1/8} < \epsilon < 2/3$. Row 1 shows the time is

$$O(\Delta n(n-\delta)^2 \log_\beta n) = O\left(\frac{n^{7/4}(n-\delta)^2}{\log\beta}\right) = O\left(\frac{n^{15/4}\epsilon^2}{\log\beta}\right).$$

Hence, it suffices to show $\ln\beta \geq \epsilon/2$ for sufficiently large n. First observe that by (10), $\beta = (n - \Delta)/\delta \geq (n - n^{3/4})/\delta = (1 - 1/n^{1/4})/(1 - \epsilon)$. Recall that $\ln(1 - x) < -x$ for $0 \leq x < 1$, and $\ln(1 - x) > -2x$ for $0 \leq x \leq 1/2$. Hence, $\ln\beta > -2/n^{1/4} + \epsilon$. For sufficiently large n, $n^{1/4} \geq 4n^{1/8}$. In this case $\epsilon > 1/n^{1/8} \geq 4/n^{1/4}$. This implies $\ln\beta \geq \epsilon/2$ as desired.

Table II row 6 shows the time for the Gap Algorithm is

$$O\left(\frac{n^{9/2}}{\Delta\log(n(H) - \delta)}\right) = O\left(\frac{n^{15/4}\log n}{\log(n(H) - \delta)}\right).$$

Hence, it suffices to show $n(H) - \delta \geq n^{3/4}$ for sufficiently large n. This follows since (10) shows $n(H) - \delta = n - \Delta - \delta > n^{7/8} - n^{3/4} \geq n^{3/4}$. $\square$

Combining Lemmas 6.1 and 6.2 gives our main result.

THEOREM 6.3. *In a digraph the vertex connectivity κ and a corresponding separator can be found in $O((n+\min\{\kappa^{5/2}, \kappa n^{3/4}\})m)$ time. In an undirected graph, the same bound holds with m replaced by κn. The space in both cases is $O(m)$.*

7. *Conclusion*

Can vertex connectivity be computed faster? We would replace the goal of linear-time connectivity computation [Aho et al. 1974] with a goal of time $O(nm)$, the same bound as the randomized algorithm of Henzinger et al. [2000]. Are expander graphs a required ingredient in efficient connectivity algorithms? For us they did the job perfectly, but perhaps other combinatorial devices can do the same or even better.

After the initial appearance of this article [Gabow 2000], an exciting breakthrough occurred, applying expander graphs to the design of efficient graph algorithms. Reingold [2005] resolved a long-standing conjecture by showing that undirected st-connectivity can be computed in logarithmic space. Reingold's approach is to transform the given graph into an expander graph, for which the problem becomes trivial.

ACKNOWLEDGMENTS. We thank Noga Alon, Avi Wigderson and David Zuckerman for help with expander graphs, and Peter Elliott for help with number theory.

REFERENCES

AHO, A., HOPCROFT, J., AND ULLMAN, J. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

AHUJA, R., MAGNANTI, T., AND ORLIN, J. 1993. *Network Flows: Theory, Algorithms and Applications*. Prentice-Hall, Englewood Cliffs, NJ.

ALON, N. 1986. Eigenvalues and expanders. *Combinatorica 6,* 2, 83–96.

ALON, N., BLUM, M., FIAT, A., KANNAN, S., NAOR, M., AND OSTROVSKY, R. 1994. Matching nuts and bolts. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, 690–696.

ALON, N., SPENCER, J., AND ERDÖS, P. 1992. *The Probabilistic Method*. Wiley Interscience, New York.

DAVIDOFF, G., SARNAK, P., AND VALETTE, A. 2003. *Elementary Number Theory, Group Theory, and Ramanujan Graphs*. London Mathematical Society Student Texts, vol. 55. Cambridge University Press, Cambridge, MA.

EVEN, S. 1975. An algorithm for determining whether the connectivity of a graph is at least k. *SIAM J. Comput. 4,* 3, 393–396.

EVEN, S., AND TARJAN, R. 1975. Network flow and testing graph connectivity. *SIAM J. Comput. 4,* 4, 507–518.

FEDER, T., AND MOTWANI, R. 1995. Clique partitions, graph compression and speeding-up algorithms. *J. Comput. System Sci. 51*, 261–272.

FRANK, A. 1994. Connectivity augmentation problems in network design. In *Mathematical Programming: State of the Art, 1994*, J. Birge and K. Murty, Eds. Univ. of Michigan Press, Ann Arbor, MI, 34–63.

GABOW, H. 2000. Using expander graphs to find vertex connectivity. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, CA, 410–420.

HENZINGER, M. 1997. A static 2-approximation algorithm for vertex connectivity and incremental approximation algorithms for edge and vertex connectivity. *J. Algorithms 24,* 1, 194–220.

HENZINGER, M., RAO, S., AND GABOW, H. 2000. Computing vertex connectivity: New bounds from old techniques. *J. Algorithms 34,* 2, 222–250.

HOORY, S., LINIAL, N., AND WIGDERSON, A. 2006. *Expander Graphs and Their Applications*. Draft.

LOVÁSZ, L. 1993. *Combinatorial Problems and Exercises, 2nd Ed.* North-Holland, Amsterdam, The Netherlands.

LUBOTZKY, A., PHILLIPS, R., AND SARNAK, P. 1988. Ramanujan graphs. *Combinatorica 8,* 3, 261–277.

MARGULIS, G. 1988. Explicit group-theoretical constructions of combinatorial schemes and their applications to the design of expanders and superconcentrators. *Problemy Peredachi Informatsii 24*, 51–60. (In Russian; English translation in *Prob. Inf. Trans. 24*, 1988, 39–46.)

MOTWANI, R., AND RAGHAVAN, P. 1995. *Randomized Algorithms*. Cambridge University Press, Cambridge, MA.

NAGAMOCHI, H., AND IBARAKI, T. 1992. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica 7*, 583–596.

REINGOLD, O. 2005. Undirected st-connectivity in log-space. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*. ACM, New York, 376–385.

SPIELMAN, D. 1995. Linear-time encodable and decodable error-correcting codes. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*. ACM, New York, 388–397.

WIGDERSON, A., AND ZUCKERMAN, D. 1993. Expanders that beat the eigenvalue bound: Explicit construction and applications. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*. ACM, New York, 245–251.

RECEIVED SEPTEMBER 2000; REVISED MARCH 2002; ACCEPTED JULY 2006