Connectivity Algorithms

Abdol–Hossein Esfahanian

Abstract

This chapter presents an exposition of the advancement of the connectivity algorithms over the years. Most of these algorithms work by making a number of *calls* to a max-flow subroutine. As these calls determine the bulk of the computation, attempts have been made to make the number of such calls as small as possible.

1. Introduction

Many algorithms for the computation of edge–connectivity and vertex–connectivity of graph and digraphs have been developed over the years. Most of these algorithms work by solving a number of *max–flow* problems (see the chapter on max–flow). In other words, these algorithms compute connectivities by making a number of *calls* to a max–flow subroutine. The major part of the computation in such algorithms comes from these calls, and as such, attempts have been made to make the number of max–flow calls as small as possible.

Even and Tarjan [6] were among the first to present max–flow based connectivity algorithms. Subsequent results include the work of Schnorr [25], Kleitman [21], Galil [10, 11], Esfahanian and Hakimi [3], Matula [23], Mansour and Schieber [22], and Henzinger and Rao [17]. The problem of determining whether λ (or κ) is larger than a prescribed value, without computing the actual value of λ (or κ), has been studied by Tarjan [26], Mansour and Schieber [23], and Gabow [9].

In this chapter, after some definitions and preliminary observations, we explain how the computation of connectivities can be reduced to solving a number of max–flow problems. We then give an exposition of the advancement of the connectivity algorithms over the years. A brief review of the literature is given in the later sections, along with some discussion. The issue of edge–connectivity will be addressed first.

2. Computing Edge–Connectivity

Let $G = (V,E)$ represent a graph (or digraph) without loops or multiple edges, with vertex set V and edge (or arc) set edge E. In a graph G, the *degree* deg(v) of a vertex v is defined as the number of edges incident to vertex v in G. The *minimum degree* $\delta(G)$ is defined as: $\delta(G) = \min\{\deg(v) \mid v \text{ in graph } G\}$. In case of a digraph, the *in–degree* in–deg(v) and the out–degree out–deg(v) are defined respectively as the number of arc incoming to and arcs outgoing from vertex v in G, and the corresponding minimum degree is: $\delta(G) = \min\{\text{in-deg}(v), \text{out-degree} \mid v \text{ in digraph } G\}$. Throughout the Chapter, we will denote the *order* and the *size* of a graph (or a digraph) by n and m, respectively.

Let v and w be a pair of distinct vertices in graph G. We define $\lambda(v, w)$ as the least number of edges whose deletion from G would destroy every path between v and w. In case of a digraph, $\lambda(v, w)$ would represent the least number of arcs whose deletion would destroy every directed path from v to w. Note that in a graph G, we have $\lambda(v, w) = \lambda(w, v)$, whereas the equality may not hold in case of a digraph.

The edge–connectivity $\lambda(G)$ of a graph G is the least cardinality $|S|$ of an edge set $S \subseteq E$ such that $G - S$ is either disconnected or trivial. Similarly, the edge–connectivity $\lambda(G)$ of a digraph G is the least cardinality $|S|$ of an arc set S such that $G - S$ is no longer strongly connected or is trivial. Such a set S is called a *minimum edge–cut* (or *arc–cut* in case of a digraph). Note that when G is not a trivial graph, we can define $\lambda(G)$ in terms of $\lambda(v, w)$ as follows: If G is a graph then

$$\lambda(G) = \min\{ \lambda(v, w) \mid \text{unordered pair } v , w \text{ in } G \} \qquad (1)$$

In case of a digraph, we have

$$\lambda(G) = \min\{ \lambda(v, w) \mid \text{ordered pair } v , w \text{ in } G \} \qquad (2)$$

The correctness of the above equalities should be clear; after all, removing a least number of edges to disconnect a graph G, for example, would in fact destroy all paths between at least a pair of vertices, and vice versa. Given the above definitions, one can compute λ of a graph (or a digraph) by knowing how to compute $\lambda(v, w)$ for arbitrary v and w.

It turns out that $\lambda(v, w)$ can be computed by solving a max–flow problem in a particular *network*, as described in the following algorithm (see Even [5]):

Algorithm 1:

Input: Graph or digraph G, and a pair of vertices v and w.
Output: Value for $\lambda(v, w)$.

1. If G is a graph, replace each edge xy with arcs (x, y) and (y, x).
2. Assign v as the *source vertex* and w as the *sink vertex*.
3. Assign the capacity of each arc to 1, and call the resulting *network H*.
4. Find a max–flow function f in H.
5. Set $\lambda(v, w)$ equal to the *total flow* of f. Stop.

As Even showed, the time complexity of the above algorithm is $O(nm)$ [5]. Provided that we have access to a max–flow software, we can use the above algorithm as a subroutine, and compute all possible $\lambda(v, w)$, take the minimum of these quantities, and subsequently compute λ. For a graph with n vertices, there are $n(n–1)/2$ unordered $\lambda(v, w)$ to compute, whereas there are $n(n–1)$ ordered $\lambda(v, w)$ to compute in case of a digraph. It turns out, however, that to determine λ, there are far fewer $\lambda(v, w)$ that we need to compute.

Consider the following abstraction of a graph G, and an arbitrary minimum edge–cut S in G (to eliminate the obvious cases in the following discussion, we will assume G is a connected nontrivial graph).

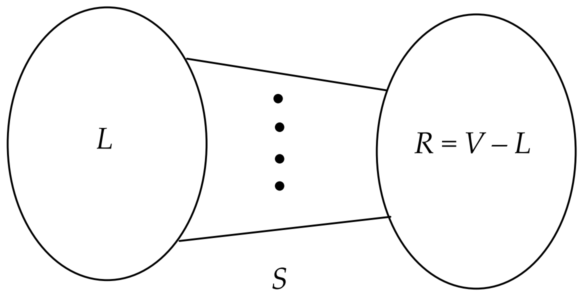

Figure 1: Graph $G = (V,E)$ and a minimum edge-cut S.

In Figure 1, L and R refer to the vertex sets of the two components of $G – S$, and we will refer to them as the "sides" of S. The key observation here is that for any vertex v in one side of S (either in L or in R) and for any vertex w in the other side (either in R or in L), we have $\lambda(v, w) = \lambda(G)$. Thus $\lambda(G)$ could be determined if we had an "oracle" as follows:

1. Select an arbitrary vertex v in some side of S.
2. Using the "oracle", identify a vertex w on the other side of S.
3. Computer $\lambda(v, w)$ using Algorithm 1. Assign $\lambda(G) \leftarrow \lambda(v, w)$. Stop.

Even and Tarjan [6] and Schnorr [25] observed that, once v is selected in the above algorithm, vertex w could be identified to within a set $X = V(G) – \{v\}$, which led to the following algorithm for computing λ.

Algorithm 2:

Input: A connected non–trivial graph $G = (V,E)$.

Output: Value of $\lambda(G)$.

1. Select an arbitrary vertex $v \in V$, and let $X = V - \{v\}$.
2. Using Algorithm 1, compute $\lambda(v, w)$ for every $w \in X$.
3. Assign $\lambda(G) \leftarrow \min\{ \lambda(v, w) \mid w \in X \}$. Stop.

The above algorithm reduces the number of computations of $\lambda(v, w)$ from $n(n-1)/2$, as discussed earlier, to $n - 1$, which is a significant reduction. If you keep staring at Figure 1, you notice that the above algorithm would correctly compute $\lambda(G)$ if set V in Step 1 is replaced by any set Y that contains vertices both from L and from R. Such a set Y will be called a *λ–covering* of G. Of course, the smaller Y, the fewer calls to the max–flow subroutine. This observation and the following lemma led to new algorithms for computing $\lambda(G)$.

It is well known that for any graph G we have $\lambda(G) \leq \delta(G)$. What happens to the size of L and R as depicted in Figure 1, when $\lambda(G) < \delta(G)$? The significant of this question will become clear later. The following lemma (see [3]) answers that question. Keep Figure 1 in mind!

Lemma 1:

Let L and R be defined as above. If $\lambda(G) < \delta(G)$ *then* $|L| > \delta$ *and* $|R| > \delta$.

Proof: Let $L = \{v_1, v_2, \dots, v_\ell\}$. We know that

$$\deg(v_1) + \deg(v_2) + \cdots + \deg(v_\ell) \geq \delta \cdot \ell.$$

We also know that

$$\deg(v_1) + \deg(v_2) + \cdots + \deg(v_\ell) = 2\,|E(< L >)| + |S|$$

where $E(< L >)$ refers to the edge set of the graph induced by the vertex set L. The right–hand side of the above equality will be maximum when L induces a complete graph, that is, each $\deg(v_i) = \ell - 1$. Thus we have

$$\delta \cdot \ell \leq \deg(v_1) + \deg(v_2) + \cdots + \deg(v_\ell) \leq \ell\,(\ell - 1) + |S|.$$

Since we are assuming that $|S| = \lambda(G) < \delta(G)$, we have $\delta \cdot \ell < \ell\,(\ell - 1) + \delta$, which implies $\ell > \delta$, if $(\ell - 1) > 0$. However, we know L contains more than one vertex because otherwise $\lambda(G)$ cannot be less than $\delta(G)$. A similar reasoning establishes that $|R| > \delta$. ■

Before discussing an application of this lemma the following observations are in order.

Corollary 1:

Let G be a graph having $\lambda(G) < \delta(G)$, *and let S be a minimum edge-cut with L and R as its sides. Then*

(a) *both L and R contain at least one vertex that is not incident to any of the edges in S.*

(b) Both L and R contain at least a non–leaf vertex of every spanning tree of G. ■

As a side note, the above corollary implies that if $\lambda(G) < \delta(G)$ then the diameter of G is at least 3.

The above corollary led to the following algorithm.

Algorithm 3:

```
Input:  A connected non–trivial graph $G = (V,E)$.
Output: Value of $\lambda(G)$.
    1. Select a spanning tree $T$ of $G$, and let $Y$ be the set of all non–leaves of $T$.
    2. Select an arbitrary vertex $v \in Y$, and let $X = Y - \{v\}$.
    3. Using Algorithm 1, compute $\lambda(v, w)$ for every $w \in X$.
    4. Assign $c \leftarrow \min\{ \lambda(v, w) \mid w \in X \}$.
    5. Assign $\lambda(G) \leftarrow \min\{ c , \delta(G) \}$. Stop.
```

The correctness of the above algorithm can been seen by noting that if $\lambda(G) < \delta(G)$ then c in Step 4 equals $\lambda(G)$, and regardless of this, Step 5 produces the correct value for λ. Note also that the more leaves T has, the fewer the calls required to Algorithm 1. However, finding a spanning tree with the maximum number of leaves is NP–hard [13]. Thus, the only savings that Algorithm 3 can guarantee is two fewer calls than Algorithm 2 would require, since any nontrivial tree has at least two leaves.

In pursuit of even smaller λ–coverings, Esfahanian and Hakimi [3] discovered that the spanning tree T produced by the following algorithm has its *leaf set* (*i.e.*, the set consisting of all leaves) as a λ–covering of G, provided that $\lambda(G) < \delta(G)$. This immediately implies by Corollary 1(b) that both L and R, as depicted in Figure 1, contain leaves as well as non–leaves of T. In other words, if Y is the set of all non–leaves of T, then both Y and $V(H) - Y$ are λ–coverings of G, provided that $\lambda(G) < \delta(G)$. For a vertex v in a graph $G = (V,E)$, we let $I(v)$ refer to the set of all edges incident at v, and $N(v)$ to refer to the set of all vertices adjacent to v.

Algorithm 4:

```
Input:  A connected non–trivial graph $G = (V,E)$
Output: Spanning tree $T = (V,E)$
    1. Assign $V(T) \leftarrow \varnothing$ and $E(T) \leftarrow \varnothing$.
```

2. Select a vertex v and assign $V(T) \leftarrow \{v\} \cup N(v)$, and $E(T) \leftarrow E(T) \cup I(v)$.
3. Select a leaf w in T such that $|N(w) \cap (V(G) - V(T))| \geq |N(r) \cap (V(G) - V(T))|$ for all leaves r in T.
4. For each neighbour u of w not in T, that is, $u \in N(w) \cap (V(G) - V(T))$, add vertex u to $V(T)$, and edge wu to $E(T)$.
5. If $|E(T)| < |V(T)| - 1$ go to Step 3. Otherwise Stop.

The essence of the above algorithm is to "grow" the partial formation of H from a leaf that contributes the most to the "growth" of T. This algorithm tends to generate spanning trees with a large number of leaves. The property of T discussed above suggests the following algorithm for computing λ.

Algorithm 5:

Input: A connected non–trivial graph $G = (V,E)$.
Output: Value of $\lambda(G)$.

1. Use Algorithm 4 to generate a spanning tree T of G. Let Y be set of all non–leaves of T. Moreover, let X be the smaller of the two sets Y and $V - Y$.
2. Select an arbitrary vertex $v \in X$, and let $Z = X - \{v\}$.
3. Using Algorithm 1, compute $\lambda(v, w)$ for every $w \in Z$.
4. Assign $c \leftarrow \min\{\lambda(v, w) \mid w \in Z\}$.
5. Assign $\lambda(G) \leftarrow \min\{c, \delta(G)\}$. Stop.

The correctness of this algorithm should be evident from the aforementioned discussion. Furthermore, since $|X| \leq n/2$, it makes no more that $n/2$ calls to Algorithm 1.

Matula [23] further improved upon Algorithm 5 by making use of Lemma 1 and dominating sets. In a graph $G = (V,E)$, a set $D \subseteq V$ is called a dominating set if for any vertex $v \in V$, either $v \in D$ or v is incident in G to a vertex in D. The following result can be easily deduced from Lemma 1.

Corollary 2.

Let G be a graph having $\lambda(G) < \delta(G)$, and let S be a minimum edge-cut with L and R as its sides. If D is a dominating set in G, then D is a λ–covering of G. ■

Corollary 2 suggests the following algorithm for computing $\lambda(G)$.

Algorithm 6.

Input: A connected non–trivial graph $G = (V,E)$.
Output: Value of $\lambda(G)$.

1. Select a dominating set D of G.

```
2. Select an arbitrary vertex $v \in D$, and let $X = D - \{v\}$.
3. Using Algorithm 1, compute $\lambda(v, w)$ for every $w \in X$.
4. Assign $c \leftarrow \min\{ \lambda(v, w) \mid w \in X \}$.
5. Assign $\lambda(G) \leftarrow \min\{ c , \delta(G) \}$. Stop.
```

Clearly this algorithm determines $\lambda(G)$ correctly. Furthermore, the smaller the set *D*, the fewer calls to Algorithm 1. While finding a smallest dominating set is NP–hard [13], finding some dominating set is easy. The next algorithm provides a way of for generating a "small" dominating set. Recall that for graph $G = (V,E)$, we define the neighbourhood set $N(X)$ of a vertex set $X \subseteq V$ as:

$$N(X) = \{ v \in V - X \mid v \text{ is adjacent in } G \text{ to some vertex in } X \}.$$

Algorithm 7:

Input: A connected non–trivial graph $G = (V,E)$.
Output: A dominating set D

```
1. Select a vertex $v \in V(D)$, and let $D = \{v\}$.
2. If $V - (D \cup N(D)) = \emptyset$, then Stop.
3. Select a vertex $w \in V - (D \cup N(D))$, and assign $D \leftarrow D \cup \{w\}$. Go to Step 2.
```

By using a dominating set *D* as produced by this algorithm, and amortizing the cost of computing $\lambda(v, w)$ for the vertices in *D*, Matula [23] was able to bring down the overall complexity of computing $\lambda(G)$ to $O(nm)$. His algorithm is the fastest known algorithm for determining $\lambda(G)$.

3. Computing Arc–Connectivity

We now turn our attention to computing arc-connectivity of digraphs. Consider the following abstraction of a digraph *D*, and an arbitrary minimum arc–cut *S* in *D* (again to eliminate the obvious cases in the following discussion, we assume *D* is a weakly connected nontrivial digraph).

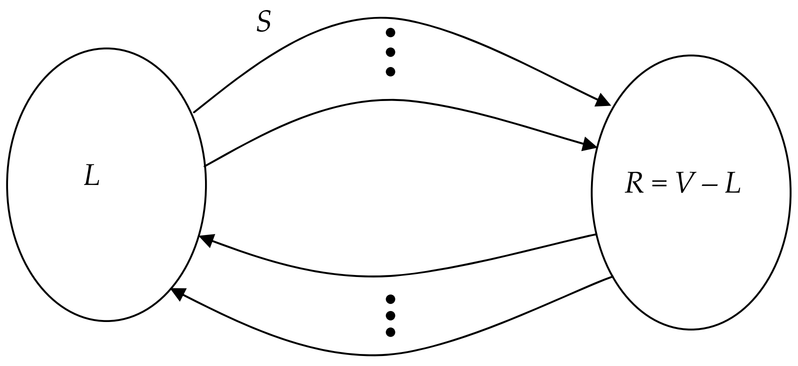

Figure 2: Digraph $D = (V,E\ )$ and a minimum arc-cut *S*.

Note that in the above abstraction, for vertices $v \in L$ and $w \in R$, we have $\lambda(D) = \lambda(v, w)$. However, it can happen that $\lambda(D) \neq \lambda(w, v)$, and for this reason, one cannot directly use Algorithm 2 to compute $\lambda(D)$ since the vertex selected in Step 1 of the algorithm may belong to R. This situation was remedied by the following lemma of Schnorr [25].

Lemma 2:

Let $Y \subseteq V(D)$ be a λ–covering for a digraph D, that is, Y contains vertices both from L and from R, as depicted in Figure 2. Further let, $Y = \{v_1, v_2, \dots, v_\ell\}$. Then

$$\lambda(D) = \min\{\, \lambda(v_1, v_2), \lambda(v_2, v_3), \lambda(v_3, v_4), \dots, \lambda(v_{\ell-1}, v_\ell), \lambda(v_\ell, v_1) \,\}.$$

Proof: Let j be the smallest index such that vertex $v_j \in Y$ is also in L; such a vertex much exist since we assume $Y \cap L \neq \varnothing$. If $j < \ell$, then let $r > j$ be the smallest index such that $v_r \in R$. In this case, we have $\lambda(D) = \lambda(v_{r-1}, v_r)$. And if $j = \ell$, let r be the smallest index such that $v_r \in R$. If $r = 1$ then we have $\lambda(G) = \lambda(v_\ell, v_r)$; otherwise we have $\lambda(D) = \lambda(v_{r-1}, v_r)$. ■

Based on this lemma, Schnorr [25] devised the following algorithm for computing arc-connectivity.

Algorithm 8:

Input: A weakly connected non–trivial digraph $D = (V,E)$.
Output: Value of $\lambda(D)$.

```
1. Let $V = \{v_1, v_2, \dots, v_n\}$.
2. Using Algorithm 1, compute $\lambda(v_1, v_2), \lambda(v_2, v_3), \lambda(v_3, v_4), \dots, \lambda(v_{n-1}, v_n)$, and
   $\lambda(v_n, v_1)$.
3. Assign $\lambda(D) = \min\{\lambda(v_1, v_2), \lambda(v_2, v_3), \lambda(v_3, v_4), \dots, \lambda(v_{n-1}, v_n), \lambda(v_n, v_1)\}$. Stop.
```

This algorithm reduces the number of calls from $n(n-1)$, as discussed earlier, to just n. Further improvements have been made based on similar techniques used in computing λ of a graph. For example, there is a version of Lemma 1 for digraphs [3]. The existence of a λ-covering Y, $|Y| \leq n/2$, when $\lambda(G) < \delta(G)$ was also shown [3]. Mansour and Schieber [22] used the notion of dominating sets, as used by Matula, and presented two algorithms for computing λ of a digraph. The combination of these algorithm yielded an $O(\min(mn, n\lambda^2))$ algorithm for computing λ of a digraph of order n and size m.

4. Computing Vertex Connectivity

In this section, we will cover some of the basic ideas in computing the vertex-connectivity of a graph; similar ideas are applicable to digraphs.

The vertex-connectivity $\kappa(G)$ of a graph $G = (V,E)$ is the least cardinality $|S|$ of a vertex set $S \subset V$ such that $G - S$ is either disconnected or trivial. Such a set S is called a *minimum vertex-cut*. We will first explain how the computation of $\kappa(G)$ reduces to solving a number of max-flow problems.

Let v and w be two vertices in graph $G = (V,E)$. If $vw \notin E$, we define $\kappa(v, w)$ as the least number of vertices, chosen from $V - \{v, w\}$, whose deletion from G would destroy every path between v and w, and if $vw \in E$ then let $\kappa(v, w) = n - 1$, where n is the order of the graph. Clearly $\kappa(G)$ can be expressed in terms of $\kappa(v, w)$ as follows:

$$\kappa(G) = \min\{ \kappa(v, w) \mid \text{unordered pair } v , w \text{ in } G \}. \tag{3}$$

It has been shown that $\kappa(v, w)$ for a pair of non-adjacent vertices v and w can be determined by solving a max-flow problem in a particular network, as described below [5]:

Algorithm 9:

Input: Graph $G = (V,E)$, and a pair of non-adjacent vertices v and w.
Output: Value for $\kappa(v, w)$.

1. Replace each edge $xy \in E$ with arcs (x, y) and (y, x), and call the resulting digraph D.
2. For each vertex u other than v and w in G, replace u with two new vertices u_1 and u_2, and then add the new arc (u_1, u_2). Connect all the arcs that were coming to u in G to u_1, and similarly, connect all the arcs that were going out of u in G to u_2 in D.
3. Assign v as the *source vertex* and w as the *sink vertex*.
4. Assign the capacity of each arc to 1, and call the resulting *network* H.
5. Find a max–flow function f in H.
6. Set $\kappa(v, w)$ equal to the *total flow* of f. Stop.

The time complexity of the above algorithm is $O(mn^{2/3})$, see [5]. Provided that one has access to a max-flow software, this algorithm can be used as a subroutine, and compute $\kappa(v, w)$ for all non-adjacent vertices v and w. This would require $n(n-1)/2 - m$ calls to Algorithm 8. However, it turns out that there are algorithms for computing κ that would require fewer calls to max-flow.

Consider the following abstraction of a graph and an arbitrary minimum vertex-cut S in G (to eliminate obvious cases in the following discussion, we will assume that G is a connected nontrivial graph, and also that G is not complete.)

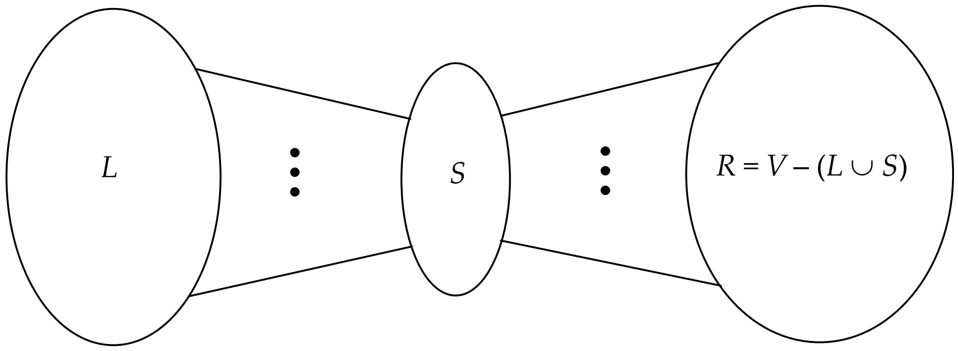

Figure 3: Graph $G = (V,E)$ and a minimum vertex-cut S

In this abstraction, L is the vertex set of one of the components of $G - S$, and R is union of the vertex sets of all the other components of $G - S$. Clearly for a vertex $v \in L$ and a vertex $w \in R$, we have $\kappa(v, w) = \kappa(G)$, and thus one might be tempted to use the same idea as in Algorithm 2, and select an arbitrary vertex v and compute

$$\min\{\ \kappa(v, w) \mid w \in V - \{v\},\ \text{and}\ w \text{ is not adjacent to } v \text{ in } G\ \}.$$

and assign it to $\kappa(G)$. However, for this computation to be true, G must have a minimum vertex-cut that does not contain v. Recall that in any graph G, we have $\kappa(G) \leq \delta(G)$. Thus, if we take a set $X \subset V$, with $|X| > \delta(G)$, then for every minimum vertex-cut S in G, there exists at least one vertex of X that is not in S, and thus κ can be computed by first finding, for each vertex $v \in X$,

$$\kappa_v = \min\{\ \kappa(v, w) \mid w \in V - \{v\},\ \text{and}\ w \text{ is not adjacent to } v \text{ in } G\ \}$$

and then setting

$$\kappa(G) = \min\{\kappa_v \mid v \in X\ \}.$$

Even and Tarjan [6] observed that if we keep track of the minimum value of $\kappa(v, w)$ as they are computed, then a set X of order $\kappa + 1$ will suffice. Here is their algorithm.

Algorithm 10:

Input: Graph $G = (V,E)$
Output: Value for κ (G).

```
1. Assign $i \leftarrow 1$, $N \leftarrow n - 1$, and let $V = \{v_1, v_2, \ldots, v_n\}$.
2. For each $j$, $j = i + 1, i + 2, \ldots, n$,
      a. If $i > N$ go to Step 4.
```

```
        b. If $v_i$ and $v_j$ are not adjacent in $G$, then compute $\kappa(v_i, v_j)$ using
           Algorithm 9, and assign $N \leftarrow \min\{N, \kappa(v_i, v_j)\}$. End of For.
    3. Assign $i \leftarrow i + 1$, and then go to Step 2.
    4. Assign $\kappa(G) \leftarrow N$, Stop.
```

This algorithm makes $O((n - \delta - 1)\kappa)$ calls to max-flow. However, the following observation of Esfahanian and Hakimi further can reduce the number of calls to max-flow for computing κ.

Take an arbitrary vertex $v \in V(G)$, and consider its situation in Figure 3. If there is a minimum vertex-cut S that does not contain v, then we have

$$\kappa(G) = \min\{\, \kappa(v, w) \mid w \in V - \{v\}, \text{ and } w \text{ is not adjacent to } v\}. \tag{4}$$

On the other hand, if v belongs to every minimum vertex-cut S, it can be shown [3] that at least two neighbours of v must lie outside S, and in this case we have

$$\kappa(G) = \min\{\, \kappa(x, y) \mid x, y \in N(v), \text{ and } x \text{ and } y \text{ are non-adjacent}\}. \tag{5}$$

Not knowing which of the above situations is true for an arbitrary vertex v, both situations must be considered. This gives the following algorithm:

Algorithm 11:

```
Input:  Graph $G = (V,E)$
Output: Value for $\kappa(G)$.
    1. Select an arbitrary vertex $v$ of minimum degree.
    2. Compute $k_1 = \min\{\kappa(v, w) \mid w \in V - \{v\}$, and $w$ is not adjacent to $v\}$.
    3. Compute $k_2 = \min\{\kappa(x, y) \mid x, y \in N(v)$, $x$ and $y$ are non-adjacent $\}$.
    4. Assign $\kappa(G) \leftarrow \min\{k_1, k_2\}$, Stop.
```

This algorithm makes $O(n - \delta - 1 + \delta(\delta - 1)/2)$ calls to max-flow. For a further refinement of this algorithm, see [3].

5. Concluding Remarks

We have covered some key developments in pursuit of fast algorithms for computing connectivities. While all these algorithms were max-flow based, researchers have tried other methods. For example, Henzinger and Rao [17] have developed a randomised algorithm for the computation of the connectivity. Algorithms have also been developed for deciding whether a graph is ℓ-edge (or ℓ-vertex) connected, some of which are max-flow based some are not. The following table gives a summary of connectivity related algorithms.

Decision	Author(s)	Year	Complexity	Comments
Edge Connectivity				
$\lambda = 2$ or $\lambda = 3$	Tarjan [26]	1972	$O(m + n)$	uses Depth First Search
λ	Even and Tarjan [6]	1975	$O(mn \times \min\{m^{1/2}, n^{2/3}\})$	n calls to max-flow
λ (digraphs)	Schnorr [25]	1979	$O(\lambda mn)$	n calls to max-flow
λ	Esfahanian & Hakimi [3]	1984	$O(\lambda mn)$	$\leq n/2$ calls to max-flow
λ (digraphs)	Esfahanian & Hakimi [3]	1984	$O(\lambda mn)$	$\leq n/2$ calls to max-flow
λ	Matula [23]	1987	$O(mn)$	uses dominating sets
$\lambda = k$	Matula [23]	1987	$O(kn^2)$	
λ (digraphs)	Mansour & Schieber [22]	1989	$O(mn)$	
$\lambda = k$	Gabow [9]	1991	$O(m+k^2 n\log(n/k))$	uses matroids
Vertex Connectivity				
$\kappa = 2$	Tarjan [26]	1972	$O(m + n)$	uses Depth First Search
$\kappa = 3$	Hopcroft & Tarjan [18]	1973	$O(m + n)$	uses triconnected components
κ	Even & Trajan [6]	1975	$O((\kappa(n - \delta - 1)mn^{2/3})$	max-flow based
$\kappa = k$	Even [4]	1975	$O(kn^3)$	max-flow based
κ	Galil [12]	1980	$O(\min\{\kappa, n^{2/3}\}mn)$	max-flow based
$\kappa = k$	Galil [12]	1980	$O(\min\{k, n^{1/2}\}kmn)$	max-flow based
κ	Esfahanian & Hakimi [3]	1984	$O((n - \delta - 1 + \delta(\delta - 1)/2)mn^{2/3})$	max-flow based
$\kappa = 4$	Kanevsky & Ramachandran [20]	1991	$O(n^2)$	
κ	Henzinger & Rao [17]	1996	$O(\kappa mn\log n)$	randomised algorithm

Table 1: A chronology of connectivity algorithms

6. References

1. M. Becker, W. Degenhardt, J. Doenhardt, S. Hertel, G. Kaninke, and W. Keber, “A probabilistic algorithm for vertex connectivity of graphs,” *Inf. Proc. Letters* 15 (1982), pp. 135-136.
2. E. A. Dinic, “Algorithm for solution of a problem of maximum flow in a network with power estimation,” *Soviet Math. Dokl.* 11 (1970), pp. 1277-1280.
3. A-H. Esfahanian and S. L. Hakimi, “On computing the connectivities of graphs and digraphs,” *NETWORKS*, Vol. 14, pp. 355-366, 1984.
4. S. Even, “An algorithm for determining whether the connectivity of a graph is at least k,” *SIAM Journal of Computing* 4 (1975), 393-396.
5. S. Even, *Graph Algorithms*, Computer Science, 1979.
6. S. Even and R. E. Tarjan, “Network flow and testing graph connectivity,” *SIAM Journal of Computing* 4 (1975), pp. 507-518.
7. H. Frank and W. Chou, “Connectivity considerations in the design of survivable networks,” *IEEE Transitions on Circuit Theory* CT-17 (1970), pp. 486-490.
8. G. N. Frederickson, “Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees,” *Journal of Computing*, Vol 26 No.2 (1997), pp. 484-538.
9. H. Gabow, “A matroid approach to finding edge connectivity and packing arborescences,” *Journal of Computer and System Science* 50(9), pp. 259-275.
10. Z. Galil, “Finding the vertex connectivity of graphs,” *SIAM Journal of Computing,* 9 (1980), pp. 197-199.
11. Z. Galil and G. F. Italiano, “Reducing edge connectivity to vertex connectivity,” *ACM SIGACT News* 22 (1991), pp. 57-61.
12. Z. Galil and G. F. Italiano, “Fully Dynamic Algorithms for 2-Edge Connectivity,” *SIAM Journal of Computing* 21 (1992), pp. 1047-1069.
13. M. R. Garey and D. S. Johnson, *Computers and Intractability, A guide to the theory of NP-Completeness*, Freeman, San Francisco (1979).
14. R. E Gomory and T.C. Hu, “Multi-terminal network flows,” *Journal of Social and Industrial and applied Mathematics* 9 (1961), pp. 551-570.
15. D. Gusfield, “Optimal Mixed Graph Augmentation,” *SIAM Journal of Computing* 16 (1987), pp. 599-612.
16. M. R. Henzinger and S. Rao, “Faster Vertex Connectivity Algorithms,” *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pp. 1-15.
17. M. R. Henzinger, S. Rao and H. N. Gabow, “Computing vertex connectivity: new bounds from old techniques,” Proc. 37th IEEE F. O. C. S. (1996), pp. 462-471.

18. J. Hopcroft, R.E. Tarjan, "Dividing a graph into triconnected components," *SIAM Journal of Computing* 2 (1973), pp. 135-158.
19. T. Hsu, *Undirected Vertex-Connectivity Structure and Smallest Four-Vertex-Connectivity Augmentation*, Nansheng University, Taiwan.
20. A. Kanevsky and V. Ramachandran, "Improved algorithms for graph four connectivity," *J. Comp. Syst. Sc*. 42 (1991), pp. 288-306.
21. D. J. Kleitman, "Methods for investigating connectivity of large graphs," *IEEE Trans. Circuit Theory* 16 (1969), pp. 232-233.
22. Y. Mansour and B. Schieber, "Finding the edge connectivity of directed graphs," *Journal of Algorithms* 10 (1989), pp. 76-85.
23. D. W. Matula, "Determining edge connectivity in *O(mn)*," *Proceedings of 28th Symp. on Foundations of Computer Science*, (1987), pp. 249-251.
24. H. Nagamochi and T. Ibaraki, "Computing edge connectivity in multigraphs and capacitated graphs," *SIAM J. Disc Math* 5 (1992), pp. 54-66.
25. C. P. Schnorr, "Bottlenecks and edge connectivity in unsymmetrical networks," *SIAM Journal of Computing* 8 (1979), 265-274.
26. R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal of Computing* 1 (1972), pp. 146-160.