On the Evolution of Graph Connectivity Algorithms

Abdol–Hossein Esfahanian
Computer Science and Engineering Department
Michigan State University
East Lansing, Michigan 48824
U.S.A.
esfahanian@cse.msu.edu

Table of Contents

1. Introduction

Many algorithms for the computation of edge–connectivity (λ) and vertex–connectivity (κ) of digraph and graphs have been developed over the years. Most of these algorithms compute λ or κ 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 is due to these calls, and as such, attempts have been made to make the number of max–flow calls as small as possible.

Even and Tarjan [7] were among the first to present max–flow based connectivity algorithms. Subsequent results include the work of Schnorr [26], Kleitman [22], Galil [11, 12], Esfahanian and Hakimi [4], Matula [24], Mansour and Schieber [23], and Henzinger and Rao [18]. 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 [28], Mansour and Schieber [24], and Gabow [10].

In this chapter, after some definitions and preliminary observation, we will first explain how the computation of connectivities can be reduced to solving a number of max–flow problems. We will 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 discussions. 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 u and v be a pair of distinct vertices in graph G. We define $\lambda(u, v)$ as the least number of edges whose deletion from G would destroy every path between u and v. In case of a digraph, $\lambda(u, v)$ would represent the least number of arcs whose deletion would destroy every directed path from u to v. Note that in a graph G, we have $\lambda(u, v) = \lambda(v, u)$, 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–separator* (or *arc–separator* in case of a digraph). Note that when G is not a trivial graph, we can define $\lambda(G)$ in terms of $\lambda(u, v)$ as follows: If G is a graph then

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

In case of a digraph, we have

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

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 $\lambda(u, v)$ for arbitrary u and v.

It turns out that $\lambda(u, v)$ can be computed by solving a max–flow problem in a particular *network*, as described in the following algorithm (see Even [6]):

Algorithm 1.

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

1. If G is a graph, replace each edge xy with arcs (x, y) and (y, x).
2. Assign u as the *source vertex* and v 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 $\lambda(u, v)$ equal to the *total flow* of f. Stop.

The time complexity of the above algorithm is $O(nm)$; see Even [6]. Provided that we have access to a max–flow software, we can use the above algorithm as a subroutine, and compute all possible $\lambda(u, v)$, take the minimum of these quantities, and subsequently compute λ. For a graph with n vertices, there are $n(n-1)/2$ unordered $\lambda(u, v)$ to compute (see relation (1)), whereas there are $n(n-1)$ ordered $\lambda(u, v)$ to compute in case of a digraph (see relation (2)). It turns out, however, that to determine λ, there are far fewer $\lambda(u, v)$ that we need to compute.

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

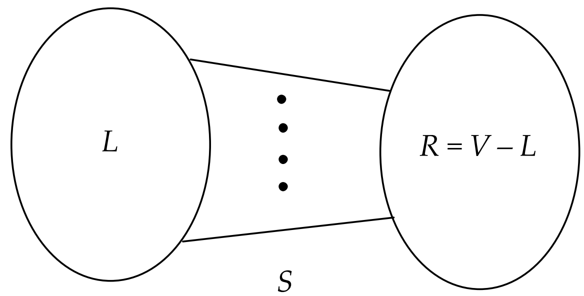

Figure 1: Graph $G = (V,E)$ and a minimum edge-separator S.

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 u in one side of S (either in L or in R) and for any vertex v in the other side (either in R or in L), we have $\lambda(u, v) = \lambda(G)$. Thus $\lambda(G)$ could be determined if we had an "oracle" as follows:

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

Even and Tarjan [7] and Schnorr [26] observed that, once u is selected in the above algorithm, vertex v could be identified to within a set $X = V(G) - \{u\}$, 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 $u \in V$, and let $X = V - \{u\}$.
2. Using Algorithm 1, compute $\lambda(u, v)$ for every $v \in X$.
3. Assign $\lambda(G) \leftarrow \min\{\lambda(u, v) \mid v \in X\}$. Stop.

The above algorithm reduces the number of computations of $\lambda(u, v)$ from $n(n-1)/2$, as discussed earlier, to $n - 1$, which is a significant reduction. If you keep staring at Figure 1, you would 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, that is, $Y \cap L \neq \emptyset$ and $Y \cap R \neq \emptyset$. 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 $\lambda(G) \le \delta(G)$. 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 [4]) answers that question. Keep Figure 1 in mind!

Lemma 1.

If $\lambda(G) < \delta(G)$ then $|L| > \delta$ and $|R| > \delta$.

Proof: Let $L = \{v_1, v_2, \ldots, v_k\}$. We know that

$$\deg(v_1) + \deg(v_2) + \cdots + \deg(v_k) \ge \delta \cdot k.$$

We also know that

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

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*. Thus we have

$$\delta \cdot k \le \deg(v_1) + \deg(v_2) + \cdots + \deg(v_k) \le 2(k(k-1)/2) + |S|.$$

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

Before discussing an application of the above lemma in computing λ, the following observations are in order.

Corollary 1.

If $\lambda(G) < \delta(G)$ then both L and R contain a vertex that is not incident to any of the edges in S. ■

As a side note, the above corollary implies that if $\lambda(G) < \delta(G)$ then the *diameter*[1] of G is at least 3.

Corollary 2.

Let T be a spanning tree of G, and let Y be the set of all non–leaf vertices of T. If $\lambda(G) < \delta(G)$ then Y is a λ–covering of G. That is, both L and R contain at least one vertex that is a non–leaf vertex in T. ■

[1] Longest shortest-path.

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–leaf vertices of T.
2. Select an arbitrary vertex $u \in Y$, and let $X = Y - \{u\}$.
3. Using Algorithm 1, compute $\lambda(u, v)$ for every $v \in X$.
4. Assign $c \leftarrow \min\{ \lambda(u, v) \mid v \in X \}$.
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 [14]. Thus, the only savings the above algorithm can guarantee is 2 fewer calls than Algorithm 2 would require, since any nontrivial tree has at least 2 leaves.

In pursuit of even smaller λ–coverings, Esfahanian and Hakimi [4] discovered that the spanning tree H produced by the following algorithm has its *leaf set* (*i.e.*, the set consisting of all leaf vertices) as a λ–covering of G, provided that $\lambda(G) < \delta(G)$. This would immediately imply by Corollary 2 that both L and R, as depicted in Figure 1, contain leaf vertices as well as non–leaf vertices of H. In other words, if we let Y be the set of all non–leaf vertices of H, then both Y and $V(H) - Y$ are λ–coverings of G, assuming that $\lambda(G) < \delta(G)$. Below is their algorithm for generating H. Given a graph $G = (V,E)$, for a vertex $u \in V(G)$, we will use $I(u)$ to refer to the set of all edges incident at u, and $A(u)$ to refer to the set of all vertices adjacent to u.

Algorithm 4.

Input: A connected non–trivial graph $G = (V,E)$
Output: Spanning tree $H = (V,E)$

1. Assign $V(H) \leftarrow \varnothing$ and $E(H) \leftarrow \varnothing$.
2. Select a vertex u and assign $V(H) \leftarrow \{u\} \cup A(u)$, and $E(H) \leftarrow E(H) \cup I(u)$.
3. Select a leaf vertex w in H such that $|A(w) \cap (V(G) - V(H))| \geq |A(r) \cap (V(G) - V(H))|$ for all leaf vertices r in H.
4. For each vertex $v \in A(w) \cap (V(G) - V(H))$, add vertex v to $V(H)$, and edge wv to $E(H)$.

5. If $|E(H)| < |V(H)| - 1$ 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 H. The above algorithm has the tendency to generate a spanning tree with a large number of leaves. The property of H 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, and generate the spanning tree H of G. Let Y be set of all non–leaf vertices of H. Moreover, let X be the smaller of the two sets Y and $V - Y$.
2. Select an arbitrary vertex $u \in X$, and let $Z = X - \{u\}$.
3. Using Algorithm 1, compute $\lambda(u, v)$ for every $v \in Z$.
4. Assign $c \leftarrow \min\{ \lambda(u, v) \mid v \in Z \}$.
5. Assign $\lambda(G) \leftarrow \min\{c, \delta(G)\}$. Stop.

The correctness of the above algorithm should be evident from the aforementioned discussions. Furthermore, since $|X| \le n/2$, where n is the order of G, the above algorithm makes no more that $n/2$ calls to Algorithm 1.

Matula [24] further improved upon the above algorithm 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 $u \in V$, either $u \in D$ or u is incident in G to a vertex in D. The following result can be easily deduced from Lemma 1.

Corollary 3.

Let D be a dominating set in G. If $\lambda(G) < \delta(G)$ then both L and R contain at least one vertex of D. That is, D is a λ–covering of G. ■

Corollary 4 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 $u \in D$, and let $X = D - \{u\}$.
3. Using Algorithm 1, compute $\lambda(u, v)$ for every $v \in X$.
4. Assign $c \leftarrow \min\{ \lambda(u, v) \mid v \in X \}$.
5. Assign $\lambda(G) \leftarrow \min\{ c, \delta(G) \}$. Stop.

Clearly the above algorithm determines $\lambda(G)$ correctly. Further, the smaller the set D, the fewer calls to Algorithm 1. While finding a least size dominating set is NP–hard [14], finding a dominating set is easy. Below is a simple algorithm for generating a "small" dominating set. In a graph $G = (V,E)$, we define the neighbourhood set $A(X)$ of a vertex set $X \subseteq V$ as:

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

Algorithm 7.

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

1. Select a vertex $u \in V(D)$, and let $D = \{u\}$.
2. If $V - (D \cup A(D)) = \varnothing$ stop.
3. Select a vertex $v \in V - (D \cup A(D))$, and assign $D \leftarrow D \cup \{v\}$. Go to Step 2.

By using a dominating set D as produced by the above algorithm, and amortizing the cost of computing $\lambda(u, v)$ for the vertices in D, Matula [24] was able to bring down the overall complexity of computing $\lambda(G)$ to $O(nm)$, where n and m are respectively the order and size of graph G. His algorithm is the fastest known algorithm for determining $\lambda(G)$.

We now turn our attention to the computation of $\lambda(G)$ when G is a digraph. Consider the following abstraction of a digraph G, and an arbitrary minimum arc–separator S in G (again to eliminate the obvious cases in the following discussion, we will assume G is a weakly–connected nontrivial graph).

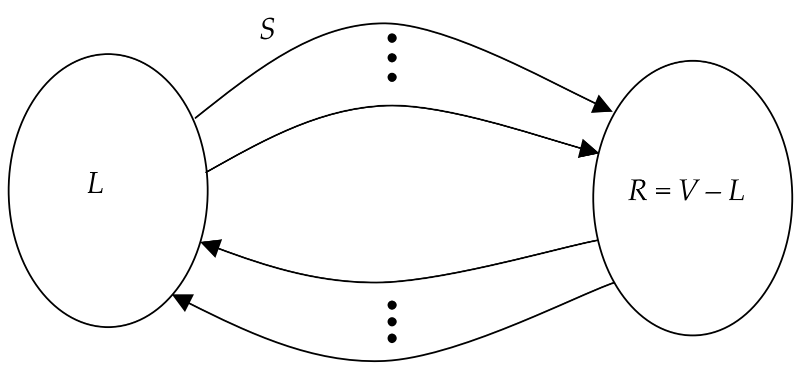

Figure 2: Digraph $G = (V,E)$ and a minimum arc-separator S.

Note that in the above abstraction, for any $u \in L$ and a vertex $v \in R$, we have $\lambda(G) = \lambda(u, v)$. However, we might have $\lambda(G) \neq \lambda(v, u)$, and for this reason, one cannot directly use Algorithm 2 to compute $\lambda(G)$ since the vertex selected in Step 1 of the algorithm may belong to R. This situation was remedied by the following lemma due to Schnorr [26].

Lemma 2.

Let $Y \subseteq V(G)$ be a λ–covering for a digraph G, that is, Y contains vertices both from L and from R, as depicted in Figure 2. Further let, $Y = \{u_1, u_2, \ldots , u_k\}$. Then

$$\lambda(G) = \min\{ \lambda(u_1, u_2), \lambda(u_2, u_3), \lambda(u_3, u_4), \ldots , \lambda(u_{k-1}, u_k), \lambda(u_k, u_1) \}.$$

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

Based on the above lemma, Schnorr [26] suggested the following algorithm for computing $\lambda(G)$ for a digraph.

Algorithm 8.

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

1. Let $V = \{u_1, u_2, \ldots , u_n\}$.
2. Using Algorithm 1, compute $\lambda(u_1, u_2), \lambda(u_2, u_3), \lambda(u_3, u_4), \ldots , \lambda(u_{n-1}, u_n)$, and $\lambda(u_n, u_1)$.
3. Assign $\lambda(G) = \min \{\lambda(u_1, u_2), \lambda(u_2, u_3), \lambda(u_3, u_4), \ldots , \lambda(u_{n-1}, u_n), \lambda(u_n, u_1)\}$.

The above algorithm reduces the number of calls from $n(n-1)$, as discussed earlier, to n. Further improvements were made based on similar techniques used in computing λ of a graph. For example, there is a version of Lemma 1 for digraphs [4]. The existence of a λ-covering Y, $|Y| \leq n/2$, when $\lambda(G) < \delta(G)$ was also shown[2] [4]. Mansour and Schieber [23] 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.

3. 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.

---

[2] Here δ is the minimum degree of the digraph.

The vertex-connectivity κ(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-separator*. We will first explain how the computation of κ(G) reduces to solving a number of max-flow problems.

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

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

It has been shown that κ(u, v) for a pair of non-adjacent vertices u and v can be determined by solving a max-flow problem in a particular network, as described below [6]:

Algorithm 9.

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

1. Replace each edge $xy \in E$ with arcs (x, y) and (y, x), and call the resulting digraph G_1.
2. For each vertex w other than u and v in G, replace w with two new vertices w_1 and w_2, and then add the new arc (w_1, w_2). Connect all the arcs that were coming to w in G to w_1, and similarly, connect all the arcs that were going out of w in G to w_2 in G_1.
3. Assign u as the *source vertex* and v 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 κ(u, v) equal to the *total flow* of f. Stop.

The time complexity of the above algorithm is $O(mn^{2/3})$, see [6]. Provided that we have access to a max-flow software, we can use the above algorithm as a subroutine, and compute κ(u, v) for all non-adjacent vertices. 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-separator S in G (to eliminate the obvious cases in the following discussion, we will assume that G is a connected nontrivial graph, and also that G is not a

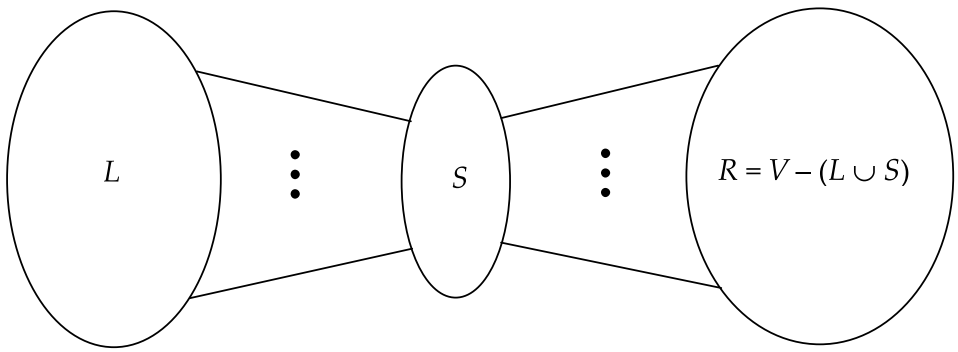

Figure 3: Graph $G = (V,E)$ and a minimum vertex-separator S

In the above 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 $u \in L$ and a vertex $v \in R$, we have $\kappa(u, v) = \kappa(G)$, and thus one might be tempted to use the same idea as in Algorithm 2, and select an arbitrary vertex u and compute

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

However, for the above relation to be true, G must have a minimum vertex-separator that does not contain vertex u. Recall that in any graph G, we have $\kappa(G) \leq \delta(G)$. Thus, if we take a set $X \subset V$, with $|X| > \delta$, then for every minimum vertex-separator S in G, there exists at least one vertex of X that is not in S, and thus κ can be computed by:

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

Even and Tarjan [7] observed that if we keep track of the minimum value of $\kappa(u, v)$ as we compute them, a set $X \subset V$, with $|X| = \kappa + 1$, would suffice. Here is their algorithm:

Algorithm 10.

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

1. Assign $i \leftarrow 1$, $N \leftarrow n - 1$, and let $V = \{u_1, u_2, \ldots, u_n\}$.
2. For each j, $j = i + 1, i + 2, \ldots, n$,
    a. If $i > N$ go to Step 4.

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

The above algorithm makes $O((n - \delta - 1)\kappa)$ calls to max-flow. However, the following observation [4] further reduces the number of calls to max-flow for computing κ.

Take an arbitrary vertex $u \in V(G)$, and let's examine its situation in Figure 2. If there is a minimum vertex-separator S which does not contain u, then we have:

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

On the other hand, if u belongs to every minimum vertex-separator S, it can be shown [4] that at least a pair of vertices adjacent to u must lie outside S, and in this case we have:

$$\kappa(G) = \min\{ \kappa(x, y) \mid x, y \in A(u), \text{ and } x \text{ and } y \text{ are non-adjacent in } G \}. \quad (5)$$

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

Algorithm 11.

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

1. Select an arbitrary vertex u of minimum degree.
2. Compute $k_1 = \min\{ \kappa(u, v) \mid v \in V - \{u\}, \text{ and } v \text{ is not adjacent to } u \text{ in } G \}$.
3. Compute $k_2 = \min\{ \kappa(x, y) \mid x, y \in A(u), x \text{ and } y \text{ are non-adjacent in } G\}$.
4. Assign $\kappa(G) \leftarrow \min\{k_1, k_2\}$, Stop.

The above algorithm makes $O(n - \delta - 1 + \delta(\delta - 1)/2)$ calls to max-flow. For a further refinement of the above algorithm see [4].

4. Concluding Remarks

We have covered some key developments in pursuit of fast algorithms for computing λ and κ. While all these algorithms were max-flow based, researchers have tried other methods. For example, Henzinger and Rao [18] have developed a randomised algorithm for the computation of κ. Algorithms have also been developed for deciding whether a graph is k-edge (or k-vertex) connected, some of which are max-flow based and some are not. The following table gives a summary of connectivity related algorithms.

Deciding	Author(s)	Year	Complexity	Comments
Edge Connectivity				
$\lambda = 2$ or $\lambda = 3$	Tarjan [28]	1972	$O(m + n)$	uses Depth First Search
λ	Even and Tarjan [7]	1975	$O(mn \times \min\{m^{1/2}, n^{2/3}\})$	n calls to max-flow
λ (digraphs)	Schnorr [26]	1979	$O(\lambda mn)$	n calls to max-flow
λ	Esfahanian & Hakimi [4]	1984	$O(\lambda mn)$	$\leq n/2$ calls to max-flow
λ (digraphs)	Esfahanian & Hakimi [4]	1984	$O(\lambda mn)$	$\leq n/2$ calls to max-flow
λ	Matula [24]	1987	$O(mn)$	uses dominating sets
$\lambda = k$	Matula [25]	1987	$O(kn^2)$	
λ (digraphs)	Mansour & Schieber [23]	1989	$O(mn)$	
$\lambda = k$	Gabow [10]	1991	$O(m+k^2n\log(n/k))$	Uses matroids
Vertex Connectivity				
$\kappa = 2$	Tarjan [28]	1972	$O(m + n)$	uses Depth First Search
$\kappa = 3$	Hopcroft & Tarjan [19]	1973	$O(m + n)$	uses triconnected components
κ	Even & Trajan [7]	1975	$O((\kappa(n - \delta - 1)mn^{2/3})$	max-flow based
$\kappa = k$	Even [5]	1975	$O(kn^3)$	max-flow based
κ	Galil [13]	1980	$O(\min\{\kappa, n^{2/3}\}mn)$	max-flow based
$\kappa = k$	Galil [13]	1980	$O(\min\{k, n^{1/2}\}kmn)$	max-flow based
κ	Becker, *et el*. [1]	1982		probabilistic algorithm
κ	Esfahanian & Hakimi [4]	1984	$O((n - \delta - 1 + \delta(\delta - 1)/2)mn^{2/3})$	max-flow based
$\kappa = 4$	Kanevsky & Ramachandran [21]	1991	$O(n^2)$	
$\kappa = k$	Cheriyan & Thurimella [3]	1991	$O(k^3n^2)$	
κ	Henzinger & Rao [18]	1996	$O(\kappa mn\log n)$	randomised algorithm

Table 1: A chronology of connectivity algorithms

5. 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), 135-136.
2. J. Cheriyan and R. Thurimella, *Algorithms for parallel k-vertex connectivity and sparse certificates*, Proceedings of the 23rd ACM Symposium on Theory of Computing, 1991.
3. E. A. Dinic, *Algorithm for solution of a problem of maximum flow in a network with power estimation*. Soviet Math. Dokl. 11 (1970), 1277-1280.
4. A. H. Esfahanian and S. L. Hakimi, *On computing the connectivities of graphs and digraphs*. Networks (1984), 355-366.
5. S. Even, *An algorithm for determining whether the connectivity of a graph is at least k*, SIAM J. Computing 4 (1975), 393-396.
6. S. Even, *Graph Algorithms*, Computer Science, Polomac, MD (1979).
7. S. Even and R. E. Tarjan, *Network flow and testing graph connectivity*, SIAM J. Computing 4 (1975), 507-518.
8. H. Frank and W. Chou, *Connectivity considerations in the design of survivable networks*. IEEE Trans. Circuit Theory CT-17 (1970), 486-490.
9. G. N. Frederickson, *Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees*, J. Comp. Vol 26 No.2 (1997), 484-538.
10. H. Gabow, *A matroid approach to finding edge connectivity and packing arborescences*, Journal of Computer and System Science 50(9): 259-- 275
11. Z. Galil, *Finding the vertex connectivity of graphs*, SIAM J. Computing 9 (1980), 197-199.
12. Z. Galil and G. F. Italiano, *Reducing edge connectivity to vertex connectivity*, ACM SIGACT News 22 (1991), 57—61.
13. Z. Galil and G. F. Italiano, *Fully Dynamic Algorithms for 2-Edge Connectivity*, SIAM J. Comput 21 (1992), 1047—1069.
14. M. R. Garey and D. S. Johnson, *Computers and Intractability, A guide to the theory of NP-Completeness*, Freeman, San Francisco (1979).
15. R. E Gomory and T.C. Hu, *Multi-terminal network flows*. J. Soc. Indust and appl. Math. 9 (1961), 551-570.
16. D. Gusfield, *Optimal Mixed Graph Augmentation*, Siam J. Computing Vol. 16 No. 4 (1987), 599—612.
17. M. R. Henzinger and S. Rao. *Faster Vertex Connectivity Algorithms. Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science.*

18. 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), 462—471.
19. J. Hopcroft, R.E. Tarjan, *Dividing a graph into triconnected components*. SIAM J. Comput 2 (1973), 135-158.
20. T. Hsu, *Undirected Vertex-Connectivity Structure and Smallest Four-Vertex-Connectivity Augmentation*, Nansheng University, Taiwan.
21. A. Kanevsky and V. Ramachandran, *Improved algorithms for graph fourconnectivity*, J. Comp. System Sci 42 (1991), 288—306.
22. D. J. Kleitman, *Methods for investigating connectivity of large graphs*, IEEE Trans. Circuit Theory CT16 (1969), 232-233.
23. Y. Mansour and B. Schieber, *Finding the edge connectivity of directed graphs*. Journal of Algorithms 10 (1989), 76-85.
24. D. W. Matula, *Determining edge connectivity in O(mn)*. Proceedings, 28th Symp. on Foundations of Computer Science, 1987 (1987), 249-251.
25. H. Nagamochi and T. Ibaraki, *Computing edge connectivity in multigraphs and capacitated graphs*, Siam J. Disc Math Vol 5 No.1 (1992), 54-66.
26. C. P. Schnorr, *Bottlenecks and edge connectivity in unsymmetrical networks*. SIAM J. Computing 8 (1979), 265-274.
27. S. Sridhar and R. Chandrasekaran, *Integer solution to synthesis of communication networks*. Integer Programming and Combinatorial Optimization. Proc. of a conference held at U of Waterloo.
28. R. E. Tarjan, *Depth first search and linear graph algorithms*. SIAM J. Comput 1 (1972), 146-160.