This chapter presents an exposition of how connectivity algorithms have advanced 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 minimize the number of such calls.

1. Introduction

A variety of algorithms for the computation of the vertex-connectivity $\kappa(G)$ and the edge-connectivity $\lambda(G)$ of a graph $G$ have been developed over the years. Most of these algorithms work by solving a number of maximum-flow problems (see Section 3 of the Preliminaries chapter); in other words, these algorithms compute the desired connectivity by making a number of calls to a max-flow subroutine. The major part of the computation in such an algorithm comes from these calls, and therefore 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 a max-flow based connectivity algorithm. 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, Rao and Gabow [17]. The problem of determining whether $\kappa(G)$ (or $\lambda(G)$) is greater than a prescribed value, without computing its actual value, has been studied by Tarjan [26], Mansour and Schieber [22] and Gabow [9].
In this chapter, 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. We begin with the edge-connectivity of graphs, then turn to the arc-connectivity of digraphs, and then the vertex-connectivity of graphs. Because the vertex-connectivity or edge-connectivity of a disconnected or trivial graph is 0, we assume throughout that our graphs are connected and non-trivial. Throughout this chapter, $n$ denotes the order of a graph or a digraph, and $m$ is the number of edges or arcs.

## 2. Computing the edge-connectivity

In this section we discuss a progression of algorithms for the edge-connectivity $\lambda(G)$ of a graph $G$. Initially, this is done via the local edge-connectivity $\lambda(v, w)$ of any pair of vertices $v$ and $w$ in $G$. By definition, $\lambda(v, w)$ is the minimum number of edges whose removal from $G$ leaves $v$ and $w$ in different components and, by Menger’s theorem, this equals the maximum number of edge-disjoint $v$–$w$ paths. The edge-connectivity of a graph $G$, being the minimum number of edges in a set whose removal leaves a disconnected graph, is clearly the least edge-connectivity of a pair of its vertices:

$$\lambda(G) = \min\{\lambda(v, w) : v, w \in V\}.$$ 

The value of $\lambda(v, w)$ can be computed by solving a max-flow problem in a network (such as that described in the Preliminaries chapter). First, convert $G$ into a network $N_G$ by replacing each edge $vw$ of $G$ by the pair of arcs $(v, w)$ and $(w, v)$, and taking the weight of each arc to be 1. Thus, we have the following algorithm (see Even [5]).

**Algorithm 1**

*Input:* A graph $G = (V, E)$ and vertices $s$ and $t$.

*Output:* $\lambda(s, t)$.

2. Assign $s$ to be the source vertex and $t$ the sink vertex.
3. Find a maximum-flow $f$ in $N_G$.
4. Set $\lambda(s, t)$ equal to the value of the flow $f$.
5. Stop.

As Even [5] showed, the time complexity of the above algorithm is $O(nm)$. Provided that we have access to max-flow software, we can use this algorithm as a subroutine to compute $\lambda(v, w)$ for all pairs of vertices $v$ and $w$, take the minimum of these quantities, and thus compute $\lambda(G)$. For a graph of order $n$, there are $\frac{1}{2} n(n-1)$ pairs of vertices for which we need to compute the edge-connectivity. It turns out, however, that we actually need to compute far fewer than this.
If $S$ is a minimum edge-cut of a graph $G$, then $G - S$ has just two components. We arbitrarily denote the two sets of vertices by $L$ and $R$ (as indicated in Fig. 1) and call them the sides of $S$. A key observation is that, for any vertices $v$ and $w$ in different sides of $S$, $\lambda(v, w) = \lambda(G)$. Thus $\lambda(G)$ can be determined if we have an oracle as follows. First, select a vertex $v$ in one side of $S$, and then, using the oracle, identify a vertex $w$ in the other side. Next, compute $\lambda(v, w)$ using Algorithm 1. By the above observation, this is $\lambda(G)$. These ideas led Even and Tarjan [6] and Schnorr [25] to construct our next algorithm.

**Algorithm 2**

**Input:** A graph $G = (V, E)$.

**Output:** $\lambda(G)$.

1. Select a vertex $v \in V$.
2. Using Algorithm 1, compute $\lambda(v, w)$ for every $w \in V - \{v\}$.
3. Assign $\lambda(G) \leftarrow \min \{\lambda(v, w)\}$.
4. Stop.

This algorithm reduces the number of computations of $\lambda(v, w)$ from our earlier value of $\frac{1}{2}n(n - 1)$ to $n - 1$, which is a significant reduction. If you keep staring at Fig. 2, you might notice that this algorithm computes $\lambda(G)$ if set $V$ in Step 1 is replaced by any set $Y$ that contains vertices from both $L$ and $R$; such a set is called a $\lambda$-covering of $G$. Formally, a subset $Y$ of the vertices in $G$, with $|Y| \geq 2$, is a $\lambda$-covering if $Y$ contains a pair of vertices $v$ and $w$, for which $\lambda(v, w) = \lambda(G)$. Clearly, the smaller the set, the fewer the calls to the max-flow subroutine. This observation and our next result led to new algorithms for computing the edge-connectivity.

It is well known that for any graph $G$, $\lambda(G) \leq \delta(G)$, the minimum vertex degree in $G$. What happens to the sizes of the sides of a minimum edge-cut (see Fig. 2) when $\lambda(G) < \delta(G)$? The significance of this question will become clear later. The following result (see [3]) answers that question. (Keep Fig. 2 in mind!)

**Theorem 2.1** Let $G$ be a graph, and let $L$ and $R$ be the two sides of a minimum edge-cut $S$. If $\lambda(G) < \delta(G)$, then $|L| > \delta(G)$ and $|R| > \delta(G)$. 
Proof. Let $L = \{v_1, v_2, \ldots, v_l\}$. On the one hand,

$$\deg v_1 + \deg v_2 + \cdots + \deg v_l \geq \delta l$$

while on the other hand,

$$\deg v_1 + \deg v_2 + \cdots + \deg v_l = 2|E(\langle L \rangle)| + |S|,$$

where $\langle L \rangle$ is the subgraph of $G$ induced by $L$. Since $\langle L \rangle$ can have no more than $\frac{1}{2}l(l-1)$ edges, and by hypothesis $|S| < \delta$, it follows that $\delta l < l(l-1) + \delta$. Since $l = 1$ gives $\lambda(G) = \delta(G)$, we see that $|L| > \delta$. The same argument applies to $R$.  

Before we discuss an application of this result, some observations are in order.

**Corollary 2.2** Let $G$ be a graph with $\lambda(G) < \delta(G)$, and let $L$ and $R$ be the sides of some minimum edge-cut $S$. Then

(a) both $L$ and $R$ contain at least one vertex that is not incident to any edge in $S$;
(b) both $L$ and $R$ contain at least one non-leaf vertex of each spanning tree of $G$.

(See Fig. 3.)

Fig. 3. A graph, a minimum edge-cut and a spanning tree (dotted edges)
Note that part (a) of the corollary implies that if $\lambda(G) < \delta(G)$, then the diameter of $G$ is at least 3. The corollary also leads to the following algorithm.

**Algorithm 3**

*Input:* A graph $G = (V, E)$.

*Output:* $\lambda(G)$.

1. Select a spanning tree $T$ of $G$, and let $Y$ be the set of non-leaves of $T$.
2. Select $v \in Y$, and let $X = Y - \{v\}$.
3. Using Algorithm 1, compute $\lambda(v, w)$ for every $w \in X$.
4. Assign $c \leftarrow \min\{\lambda(v, w) : w \in X\}$.
5. Assign $\lambda(G) \leftarrow \min\{c, \delta(G)\}$.
6. Stop.

The correctness of this algorithm can be 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 $\lambda(G)$. Note also that the more leaves $T$ has, the fewer are the calls required to Algorithm 1. However, as shown by Garey and Johnson [13], finding a spanning tree with the maximum number of leaves is NP-hard. Thus, the only savings that Algorithm 3 can guarantee is two fewer calls than Algorithm 2, since every non-trivial tree has at least two leaves.

In pursuit of even smaller $\lambda$-coverings, Esfahanian and Hakimi [3] discovered that the set of leaves of the spanning tree $T$ produced by the next algorithm is a $\lambda$-covering of $G$, provided that $\lambda(G) < \delta(G)$. By part (b) of Corollary 2.2, this immediately implies that both $L$ and $R$ (as indicated in Fig. 2) contain both leaves and non-leaves of $T$. In other words, if $Y$ is the set of all non-leaves of $T$, then both $Y$ and $V - Y$ are $\lambda$-coverings of $G$, provided that $\lambda(G) < \delta(G)$. (For each vertex $v$ in a graph $G$, $E(v)$ denotes the set of edges incident with $v$, and $N(v)$ denotes the set of neighbours of $v$.)

**Algorithm 4**

*Input:* A graph $G = (V, E)$.

*Output:* A spanning tree $T$.

1. Assign $V(T) \leftarrow \{\}$ and $E(T) \leftarrow \{\}$.
2. Select a vertex $v \in V$, and assign $V(T) \leftarrow \{v\} \cup N(v)$ and $E(T) \leftarrow E(v)$.
3. Select a leaf $w$ in $T$ for which $|N(w) \cap (V(G) - V(T))|$ is maximum.
4. For each neighbour $u$ of $w$ not in $T$, add $u$ to $V(T)$ and $uw$ to $E(T)$.
5. If $|E(T)| < |V(T)| - 1$, go to Step 3.
6. 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 $\lambda(G)$. 
Algorithm 5
Input: A graph \(G = (V, E)\).
Output: \(\lambda(G)\).

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

The correctness of this algorithm should be evident from the discussion. Furthermore, since \(|X| \leq \frac{1}{2} n\), it makes at most \(\frac{1}{2} n\) calls to Algorithm 1.

Matula [23] improved further upon Algorithm 5 by making use of Theorem 2.1 and dominating sets; recall that a dominating set in a graph is a set \(X\) of vertices with the property that each vertex in the graph is either in \(X\) or adjacent to a vertex in \(X\).

The following result can be easily deduced from Theorem 1.2.

**Corollary 2.3** Let \(S\) be a minimum edge-cut in a graph \(G\) with \(\lambda(G) < \delta(G)\). Then every dominating set in \(G\) is a \(\lambda\)-covering of \(G\).

This corollary suggests the following algorithm for computing the edge-connectivity.

Algorithm 6
Input: A graph \(G = (V, E)\).
Output: \(\lambda(G)\).

1. Select a dominating set \(X\) of \(G\).
2. Select an arbitrary vertex \(v \in X\), and let \(W = X - \{v\}\).
3. Using Algorithm 1, compute \(\lambda(v, w)\) for every \(w \in W\).
4. Assign \(c \leftarrow \min\{\lambda(v, w) : w \in W\}\).
5. Assign \(\lambda(G) \leftarrow \min\{c, \delta(G)\}\).
6. Stop.

Clearly, this algorithm determines \(\lambda(G)\) correctly. Furthermore, the smaller the dominating set, the fewer are the calls to Algorithm 1. While finding a smallest dominating set is NP-hard (see, for example, Garey and Johnson [13]), finding some dominating set is easy. The next algorithm provides a way of generating a ‘small’ dominating set; recall that the neighbourhood \(N(X)\) of a set \(X\) of vertices is the union of the neighbourhoods of the vertices in \(X\).

Algorithm 7
Input: A connected non-trivial graph \(G = (V, E)\).
Output: A dominating set \(X\).
1. Select a vertex \( v \in V \), and let \( X = \{v\} \).
2. If \( V - (X \cup N(X)) = \{\} \) then Stop.
3. Select \( w \in V - (X \cup N(X)) \).
4. Assign \( X \leftarrow X \cup \{w\} \).
5. Go to Step 2.

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

### 3. Computing the arc-connectivity

We now turn our attention to computing the arc-connectivity of digraphs, and for simplicity we assume that \( D \) is a strongly connected and non-trivial digraph. Consider the representation of such a digraph \( D \) in Fig. 4, with an arbitrary minimum arc-cut \( S \) of minimum size.

![Fig. 4. A digraph, minimum arc-cut \( S \), and its \( L \) and \( R \) sides](image)

Note that, for any vertices \( v \in L \) and \( w \in R \), \( \lambda(D) = \lambda(v, w) \). However, for general vertices \( v \) and \( w \), it may be that \( \lambda(D) \neq \lambda(w, v) \). Consequently, one cannot use Algorithm 2 directly 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 result of Schnorr [25].

**Theorem 3.1** Let \( D \) be a digraph, \( S \) be a minimum arc-cut with left and right sides \( L \) and \( R \) and \( W = \{w_1, w_2, \ldots, w_l\} \) be a \( \lambda \)-covering of \( D \) for \( S \). Then

\[
\lambda(D) = \min\{\lambda(w_1, w_2), \lambda(w_2, w_3), \ldots, \lambda(w_{l-1}, w_l), \lambda(w_l, w_1)\}.
\]
Clearly, we may assume that the vertices in $W$ are ordered so that $w_1 \in L$. Let $r$ be the smallest index for which vertex $w_r$ is in $R$; such a vertex must exist, since $W$ is a $\lambda$-covering. It follows that $\lambda(D) = \lambda(w_{r-1}, w_r)$. ■

Based on this result, Schnorr [25] devised the following algorithm for computing the arc-connectivity of a digraph.

**Algorithm 8**

*Input*: A strongly connected digraph $D = (V, E)$.

*Output*: $\lambda(D)$.

1. Let $V = \{v_1, v_2, \ldots, v_n\}$.
2. Using Algorithm 1, compute $\lambda(v_i, v_{i+1})$ for $i = 1, 2, \ldots, n-1$, and $\lambda(v_n, v_1)$.
3. Assign $\lambda(D) \leftarrow \min\{\lambda(v_1, v_2), \lambda(v_2, v_3), \ldots, \lambda(v_{n-1}, v_n), \lambda(v_n, v_1)\}$.
4. 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 techniques similar to others used in computing the edge-connectivity of a graph; for example, there is a version of Theorem 2.1 for digraphs [3]. The existence of a $\lambda$-covering with at most half the vertices was also shown for every graph $G$ for which $\lambda(G) < \delta(G)$ [3]. Mansour and Schieber [22] used the notion of dominating sets (as developed by Matula) to create two algorithms for computing the arc-connectivity. A combination of their algorithms yields one of order $O(\min\{mn, n\lambda^2\})$.

### 4. Computing the vertex-connectivity

In this section we cover some of the basic ideas involved in computing the connectivity (also called the vertex-connectivity) of a graph. Because the ideas are so similar for digraphs, we focus on the undirected case here. We begin by explaining how the computation of the connectivity can be reduced to solving a number of maximum-flow problems, which are discussed in the Preliminaries chapter.

Recall that $\kappa(G)$ is the minimum number of vertices whose removal from graph $G$ leaves a disconnected or a trivial graph, while for non-adjacent vertices $v$ and $w$, $\kappa(v, w)$ is the minimum number of vertices whose removal leaves a graph in which $v$ and $w$ lie in different components. (See Chapter 1 for a survey of graph connectivity.) Just as there is a connection between global and local edge-connectivities,

$$\lambda(G) = \min\{\lambda(v, w) : v, w \in V\},$$

so there is a similar one for vertex-connectivities,

$$\kappa(G) = \min\{\kappa(v, w) : v, w \in V\},$$

provided that when $v$ and $w$ are adjacent, we define $\kappa(v, w)$ in $G$ to be one more than its value in $G - vw$. To avoid trivial cases, we consider only non-complete but connected graphs.
Even [5] showed that, for non-adjacent vertices \( v \) and \( w \), the local connectivity \( \kappa(v, w) \) can be found by solving a maximum-flow problem in a related network. Given a graph \( G \) and a pair \( s \) and \( t \) of non-adjacent vertices, let the network \( N_G \) be defined as follows. First, replace each edge \( vw \) of \( G \) by an arc from \( v \) to \( w \) and an arc from \( w \) to \( v \) to form the digraph \( D_G \); arcs coming into \( s \) or going away from \( t \) should be removed, as they play no role in finding a max-flow. Then form \( N_G \) from \( D_G \) by successively replacing each vertex \( v_i \) other than \( s \) and \( t \) by a pair of vertices \( u_i \) and \( w_i \), adding an arc from \( u_i \) to \( w_i \), and replacing each arc into \( v_i \) with an arc into \( u_i \) and each arc out of \( v_i \) with an arc out of \( w_i \) (with the corresponding other ends). Finally, assign each arc weight 1. An example is shown in Fig. 5.

![Diagram](image)

Fig. 5. A graph \( G \), its digraph \( D_G \) and its associated network \( N_G \)

Here is Even’s algorithm.

**Algorithm 9**

*Input:* A graph \( G = (V, E) \) with non-adjacent vertices \( s \) and \( t \).

*Output:* \( \kappa(s, t) \).

1. Form the associated network \( N_G \).
2. Find a maximum-flow function \( f \) for \( N_G \).
3. Set \( \kappa(s, t) \) equal to the value of \( f \).
4. Stop.
Even [5] showed that the time complexity of his algorithm is $O(mn^{2/3})$. The algorithm can be used as a subroutine to compute $\kappa(v, w)$ for all pairs of non-adjacent vertices $v$ and $w$. Just as Algorithm 1, a local result, leads to global versions (such as Algorithm 2), so Algorithm 9 leads to an analogous global algorithm.

**Algorithm 10**  
*Input:* A graph $G = (V, E)$.  
*Output:* $\kappa(G)$.

1. Using Algorithm 9, compute $\kappa(v, w)$ for every pair of non-adjacent vertices $v$ and $w$.  
2. Assign $\kappa(G) \leftarrow \min\{\kappa(v, w)\}$.  
3. Stop.

This algorithm requires $\frac{1}{2} n(n-1) - m$ calls to Algorithm 9. However, there are algorithms for computing $\kappa$ that require fewer calls to a maximum-flow algorithm, one of which we next describe.

Consider the representation of a graph $G$ shown in Fig. 6, where $S$ is a minimum cutset. (As before, we assume that $G$ is connected and not complete.) Further, $L$ is the set of vertices in one component of $G - S$ and $R$ consists of the remaining vertices. Clearly, for vertices $v \in L$ and $w \in R$, $\kappa(v, w) = \kappa(G)$, and so one might be tempted to use the same idea as was used in Algorithm 2, by choosing an arbitrary vertex $v$ and finding the minimum connectivity between $v$ and a vertex not adjacent to it.

However, for this to work, there must be a minimum cutset that does not contain $v$. This can be achieved in the following way. Recall that in any graph $G$, $\kappa(G) \leq \delta(G)$. Hence, if $S$ is a minimum cutset of $G$ and $X$ is any set of more than $\delta(G)$ vertices, then $X$ has at least one vertex that is not in $S$. Therefore, $\kappa(G)$ can be computed as

$$\kappa(G) = \min_{v \in X} \{\min\{\kappa(v, w) : w \neq v\}\}.$$
Even and Tarjan [6] observed that, if we keep track of the minimum of the connectivities \( \kappa(v, w) \) as these values are computed, then a set \( X \) of order \( \kappa(G) + 1 \) suffices. Here is their algorithm.

**Algorithm 11**

*Input:* A graph \( G = (V, E) \).

*Output:* \( \kappa(G) \).

1. Assign \( i \leftarrow 1, k \leftarrow n - 1, \) and let \( V = \{v_1, v_2, \ldots, v_n\} \).
2. For \( j = i + 1, i + 2, \ldots, n, \)
   2.1 If \( i > k \), go to Step 4;
   2.2 If \( v_i \) and \( v_j \) are not adjacent, compute \( \kappa(v_i, v_j) \) using Algorithm 9, and assign \( k \leftarrow \min\{k, \kappa(v_i, v_j)\} \).
3. Assign \( i \leftarrow i + 1, \) and go to Step 2.
4. Assign \( \kappa(G) \leftarrow k \).
5. Stop.

This algorithm makes \( O((n - \delta(G) - 1)\kappa(G)) \) calls to max-flow. However, Esfahanian and Hakimi [3] observed that the number of calls can be reduced further.

Take an arbitrary vertex \( v \), and consider again the situation indicated in Fig. 6. If there is a minimum cutset \( S \) that does not contain \( v \), then

\[
\kappa(G) = \min\{\kappa(v, w) : w \neq v\}.
\]

On the other hand, if \( v \) belongs to every minimum cutset of \( G \), then it can be shown (see [3]) that at least two neighbours of \( v \) are not in \( S \), and so in this case,

\[
\kappa(G) = \min\{\kappa(u, w) : u, w \in N(v), u \neq w\}.
\]

Since we do not know which of these situations applies to our arbitrary vertex \( v \), both must be considered. This yields our next algorithm.

**Algorithm 12**

*Input:* A graph \( G = (V, E) \).

*Output:* \( \kappa(G) \).

1. Select a vertex \( v \) of minimum degree.
2. Compute \( k_1 = \min\{\kappa(v, w) : w \neq v\} \).
3. Compute \( k_2 = \min\{\kappa(u, w) : u, w \in N(v), u \neq w\} \).
4. Assign \( \kappa(G) \leftarrow \min\{k_1, k_2\} \).
5. Stop.

This algorithm makes \( O(n + \delta^2) \) calls to the maximum-flow algorithm; for a further refinement, see [3].
5. Concluding remarks

In this chapter, we have presented some of the key developments that have arisen in the pursuit of fast algorithms for computing graph connectivities. All of these algorithms are max-flow-based, but researchers have tried other methods as well. For example, Henzinger and Rao [16] developed a randomized algorithm for computing the connectivity. Algorithms have also been developed for deciding whether a graph is $k$-connected or $l$-edge-connected, some of which are not max-flow based. We conclude with a table that summarizes facts (for a graph $G$ with $n$ vertices and $m$ edges) concerning algorithms related to connectivity.

<table>
<thead>
<tr>
<th>Decision</th>
<th>Authors</th>
<th>Year</th>
<th>Complexity</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge-connectivity or Arc-connectivity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\lambda = 2, 3$</td>
<td>Tarjan [26]</td>
<td>1972</td>
<td>$O(m + n)$</td>
<td>uses depth-first search</td>
</tr>
<tr>
<td>$\lambda$ (general)</td>
<td>Even and Tarjan [6]</td>
<td>1975</td>
<td>$O(\min{m^{3/2}n, mn^{5/3}})$</td>
<td>$n$ calls to max-flow</td>
</tr>
<tr>
<td>$\lambda$ (digraph)</td>
<td>Schnorr [25]</td>
<td>1979</td>
<td>$O(\lambda mn)$</td>
<td>$n$ calls to max-flow</td>
</tr>
<tr>
<td>$\lambda$ (general)</td>
<td>Esfahanian and Hakimi [3]</td>
<td>1984</td>
<td>$O(\lambda mn)$</td>
<td>at most $\frac{1}{2}n$ calls to max-flow</td>
</tr>
<tr>
<td>$\lambda$ (digraph)</td>
<td>Esfahanian and Hakimi [3]</td>
<td>1984</td>
<td>$O(\lambda mn)$</td>
<td>at most $\frac{1}{2}n$ calls to max-flow</td>
</tr>
<tr>
<td>$\lambda$ (general)</td>
<td>Matula [23]</td>
<td>1987</td>
<td>$O(mn)$</td>
<td>uses dominating sets</td>
</tr>
<tr>
<td>$\lambda = l$</td>
<td>Matula [23]</td>
<td>1987</td>
<td>$O(ln^2)$</td>
<td></td>
</tr>
<tr>
<td>$\lambda$ (digraph)</td>
<td>Mansour and Schieber [22]</td>
<td>1989</td>
<td>$O(mn)$</td>
<td></td>
</tr>
<tr>
<td>$\lambda = l$</td>
<td>Gabow [9]</td>
<td>1991</td>
<td>$O(m + l^2n \log(n/l))$</td>
<td>uses matroids</td>
</tr>
<tr>
<td>Vertex-connectivity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\kappa = 2$</td>
<td>Tarjan [26]</td>
<td>1972</td>
<td>$O(m + n)$</td>
<td>uses depth-first search</td>
</tr>
<tr>
<td>$\kappa = 3$</td>
<td>Hopcroft and Tarjan [18]</td>
<td>1973</td>
<td>$O(m + n)$</td>
<td>uses 3-connected components</td>
</tr>
<tr>
<td>$\kappa$ (general)</td>
<td>Even and Tarjan [6]</td>
<td>1975</td>
<td>$O(\kappa(n - \delta - 1)mn^{2/3})$</td>
<td>max-flow based</td>
</tr>
<tr>
<td>$\kappa = k$</td>
<td>Even [4]</td>
<td>1975</td>
<td>$O(kn^3)$</td>
<td>max-flow based</td>
</tr>
<tr>
<td>$\kappa$ (general)</td>
<td>Galil [10]</td>
<td>1980</td>
<td>$O(mn \min{\kappa, n^{2/3}})$</td>
<td>max-flow based</td>
</tr>
<tr>
<td>$\kappa = k$</td>
<td>Galil [10]</td>
<td>1980</td>
<td>$O(kmn \min{k, n^{1/2}})$</td>
<td>max-flow based</td>
</tr>
<tr>
<td>$\kappa$ (general)</td>
<td>Esfahanian and Hakimi [3]</td>
<td>1984</td>
<td>$O((n - 1 + \frac{1}{2}\delta(\delta - 3)mn^{2/3})$</td>
<td>max-flow based</td>
</tr>
<tr>
<td>$\kappa = 4$</td>
<td>Kanevsky and Ramachandran [20]</td>
<td>1991</td>
<td>$O(n^5)$</td>
<td></td>
</tr>
<tr>
<td>$\kappa$ (general)</td>
<td>Henzinger and Rao [16]</td>
<td>1996</td>
<td>$O(kmn \log n)$</td>
<td>randomized algorithm</td>
</tr>
</tbody>
</table>
References

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