7 Connectivity

Frank Kammer and Hanjo Täubig

This chapter is mainly concerned with the strength of connections between vertices with respect to the number of vertex- or edge-disjoint paths. As we shall see, this is equivalent to the question of how many nodes or edges must be removed from a graph to destroy all paths between two (arbitrary or specified) vertices. For basic definitions of connectivity see Section 2.2.4.

We present algorithms which

- check k-vertex (k-edge) connectivity,
- compute the vertex (edge) connectivity, and
- compute the maximal k-connected components

of a given graph.

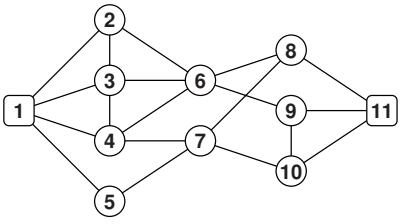
After a few definitions we present some important theorems which summarize fundamental properties of connectivity and which provide a basis for understanding the algorithms in the subsequent sections.

We denote the vertex-connectivity of a graph G by $\kappa(G)$ and the edge-connectivity by $\lambda(G)$; compare Section 2.2.4. Furthermore, we define the local (vertex-)connectivity $\kappa_G(s,t)$ for two distinct vertices s and t as the minimum number of vertices which must be removed to destroy all paths from s to t. In the case that an edge from s to t exists we set $\kappa_G(s,t) = n-1$ since κ_G cannot exceed $n-2$ in the other case[1]. Accordingly, we define $\lambda_G(s,t)$ to be the least number of edges to be removed such that no path from s to t remains. Note, that for undirected graphs $\kappa_G(s,t) = \kappa_G(t,s)$ and $\lambda_G(s,t) = \lambda_G(t,s)$, whereas for directed graphs these functions are, in general, not symmetric.

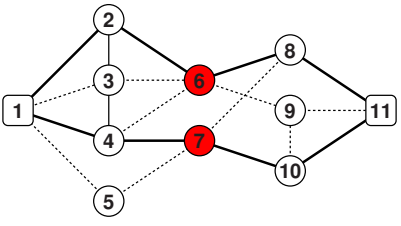
Some of the terms we use in this chapter occur under different names in the literature. In what follows, we mainly use (alternatives in parentheses): cut-vertex (articulation point, separation vertex), cut-edge (isthmus, bridge), component (connected component), biconnected component (non-separable component, block). A *cut-vertex* is a vertex which increases the number of connected components when it is removed from the graph; the term *cut-edge* is defined similarly. A *biconnected component* is a maximal 2-connected subgraph; see Chapter 2. A *block* of a graph G is a maximal connected subgraph of G containing no cut-vertex, that is, the set of all blocks of a graph consists of its isolated

[1] If s and t are connected by an edge, it is not possible to disconnect s from t by removing only vertices.

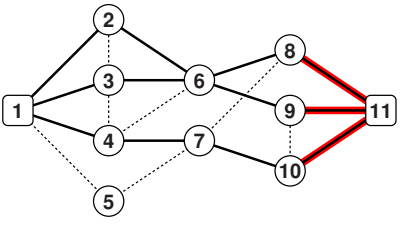
U. Brandes and T. Erlebach (Eds.): Network Analysis, LNCS 3418, pp. 143–177, 2005.



(a) A graph. We consider the connectivity between the vertices 1 and 11.



(b) 2 vertex-disjoint paths and a vertex-cutset of size 2.



(c) 3 edge-disjoint paths and an edge-cutset of size 3.

Fig. 7.1. Vertex-/edge-disjoint paths and vertex-/edge-cutsets

vertices, its cut-edges, and its maximal biconnected subgraphs. Hence, with our definition, a block is (slightly) different from a biconnected component.

The *block-graph* $B(G)$ of a graph G consists of one vertex for each block of G. Two vertices of the block-graph are adjacent if and only if the corresponding blocks share a common vertex (that is, a cut-vertex). The *cutpoint-graph* $C(G)$ of G consists of one vertex for each cut-vertex of G, where vertices are adjacent if and only if the corresponding cut-vertices reside in the same block of G. For the block- and the cutpoint-graph of G the equalities $B(B(G)) = C(G)$ and $B(C(G)) = C(B(G))$ hold [275]. The *block-cutpoint-graph* of a graph G is the bipartite graph which consists of the set of cut-vertices of G and a set of vertices which represent the blocks of G. A cut-vertex is adjacent to a block-vertex whenever the cut-vertex belongs to the corresponding block. The block-cutpoint-graph of a connected graph is a tree [283]. The maximal k-vertex-connected (k-edge-connected) subgraphs are called *k-vertex-components* (*k-edge-components*). A k-edge-component which does not contain any $(k+1)$-components is called a *cluster* [410, 470, 411, 412].

7.1 Fundamental Theorems

Theorem 7.1.1. *For all non-trivial graphs G it holds that:*

$$\kappa(G) \leq \lambda(G) \leq \delta(G)$$

Proof. The incident edges of a vertex having minimum degree $\delta(G)$ form an edge separator. Thus we conclude $\lambda(G) \leq \delta(G)$.

The vertex-connectivity of any graph on n vertices can be bounded from above by the connectivity of the complete graph $\kappa(K_n) = n - 1$.

Let $G = (V, E)$ be a graph with at least 2 vertices and consider a minimal edge separator that separates a vertex set S from all other vertices $\bar{S} = V \setminus S$. In the case that all edges between S and $\bar{S}$ are present in G we get $\lambda(G) = |S| \cdot |\bar{S}| \geq |V| - 1$. Otherwise there exist vertices $x \in S, y \in \bar{S}$ such that $\{x, y\} \notin E$, and the set of all neighbors of x in $\bar{S}$ as well as all vertices from $S \setminus \{x\}$ that have neighbors in $\bar{S}$ form a vertex separator; the size of that separator is at most the number of edges from S to $\bar{S}$, and it separates (at least) x and y. □

The following is the graph-theoretic equivalent of a theorem that was published by Karl Menger in his work on the general curve theory [419].

Theorem 7.1.2 (Menger, 1927). *If P and Q are subsets of vertices of an undirected graph, then the maximum number of vertex-disjoint paths connecting vertices from P and Q is equal to the minimum cardinality of any set of vertices intersecting every path from a vertex in P to a vertex in Q.*

This theorem is also known as the n-chain or n-arc theorem, and it yields as a consequence one of the most fundamental statements of graph theory:

Corollary 7.1.3 (Menger's Theorem). *Let s, t be two vertices of an undirected graph $G = (V, E)$. If s and t are not adjacent, the maximum number of vertex-disjoint s-t-paths is equal to the minimum cardinality of an s-t-vertex-separator.*

The analog for the case of edge-cuts is stated in the next theorem.

Theorem 7.1.4. *The maximum number of edge-disjoint s-t-paths is equal to the minimum cardinality of an s-t-edge-separator.*

This theorem is most often called the edge version of Menger's Theorem although it was first explicitely stated three decades after Menger's paper in publications due to Ford and Fulkerson [218], Dantzig and Fulkerson [141], as well as Elias, Feinstein, and Shannon [175].

A closely related result is the Max-Flow Min-Cut Theorem by Ford and Fulkerson (see Theorem 2.2.1, [218]). The edge variant of Menger's Theorem can be seen as a restricted version where all edge capacities have a unit value.

The following global version of Menger's Theorem was published by Hassler Whitney [581] and is sometimes referred to as 'Whitney's Theorem'.

Theorem 7.1.5 (Whitney, 1932). *Let $G = (V, E)$ be a non-trivial graph and k a positive integer. G is k-(vertex-)connected if and only if all pairs of distinct vertices can be connected by k vertex-disjoint paths.*

The difficulty in deriving this theorem is that Menger's Theorem requires the nodes to be not adjacent. Since this precondition is not present in the edge version of Menger's Theorem, the following follows immediately from Theorem 7.1.4:

Theorem 7.1.6. *Let $G = (V, E)$ be a non-trivial graph and k a positive integer. G is k-edge-connected if and only if all pairs of distinct vertices can be connected by k edge-disjoint paths.*

For a detailed review of the history of Menger's Theorem we refer to the survey by Schrijver [506].

Beineke and Harary discovered a similar theorem for a combined vertex-edge-connectivity (see [55]). They considered *connectivity pairs* (k, l) such that there is some set of k vertices and l edges whose removal disconnects the graph, whereas there is no set of $k-1$ vertices and l edges or of k vertices and $l-1$ edges forming a mixed vertex/edge cut set.

Theorem 7.1.7 (Beineke & Harary, 1967). *If (k, l) is a connectivity pair for vertices s and t in graph G, then there are $k + l$ edge-disjoint paths joining s and t, of which k are mutually non-intersecting.*

The following theorem gives bounds on vertex- and edge-connectivity (see [274]).

Theorem 7.1.8. *The maximum (vertex-/edge-) connectivity of some graph on n vertices and m edges is*

$$\begin{cases} \left\lfloor \frac{2m}{n} \right\rfloor & , \text{ if } m \geq n-1 \\ 0 & , \text{ otherwise.} \end{cases}$$

The minimum (vertex-/edge-) connectivity of some graph on n vertices and m edges is

$$\begin{cases} m - \binom{n-1}{2} & , \text{ if } \binom{n-1}{2} < m \leq \binom{n}{2} \\ 0 & , \text{ otherwise.} \end{cases}$$

A further proposition concerning the edge connectivity in a special case has been given by Chartrand [114]:

Theorem 7.1.9. *For all graphs $G = (V, E)$ having minimum degree $\delta(G) \geq \lfloor |V|/2 \rfloor$, the edge-connectivity equals the minimum degree of the graph: $\lambda(G) = \delta(G)$*

For more bounds on graph connectivity see [28, 62, 390, 63, 182, 523].

The following theorems deal with the k-vertex/edge-components of graphs. The rather obvious facts that two different components of a graph have no vertex in common, and two different blocks share at most one common vertex, have been generalized by Harary and Kodama [279]:

Theorem 7.1.10. *Two distinct k-(vertex-)components have at most $k-1$ vertices in common.*

While k-vertex-components might overlap, k-edge-components do not.

Theorem 7.1.11 (Matula, 1968). *For any fixed natural number $k \geq 1$ the k-edge-components of a graph are vertex-disjoint.*

Proof. The proof is due to Matula (see [410]). Consider an (overlapping) decomposition $\tilde{G} = G_1 \cup G_2 \cup \ldots \cup G_t$ of a connected subgraph $\tilde{G}$ of G. Let $C = (A, \bar{A})$ be a minimum edge-cut of $\tilde{G}$ into the disconnected parts A and $\bar{A}$. To skip the trivial

case, assume that $\tilde{G}$ has at least 2 vertices. For each subgraph G_i that contains a certain edge $e \in C$ of the min-cut, the cut also contains a cut for G_i (otherwise the two vertices would be connected in $G_i \setminus C$ and $\tilde{G} \setminus C$ which would contradict the assumption that C is a minimum cut). We conclude that there is a G_i such that $\lambda(\tilde{G}) = |C| \geq \lambda(G_i)$, which directly implies $\lambda(\tilde{G}) \geq \min_{1 \leq i \leq t}\{\lambda(G_i)\}$ and thereby proves the theorem. □

Although we can see from Theorem 7.1.1 that k-vertex/edge-connectivity implies a minimum degree of at least k, the converse is not true. But in the case of a large minimum degree, there must be a highly connected subgraph.

Theorem 7.1.12 (Mader, 1972). *Every graph of average degree at least $4k$ has a k-connected subgraph.*

For a proof see [404].

Several observations regarding the connectivity of *directed* graphs have been made. One of them considers directed spanning trees rooted at a node r, so called *r-branchings*:

Theorem 7.1.13 (Edmonds' Branching Theorem [171]). *In a directed multigraph $G = (V, E)$ containing a vertex r, the maximum number of pairwise edge-disjoint r-branchings is equal to $\kappa_G(r)$, where $\kappa_G(r)$ denotes the minimum, taken over all vertex sets $S \subset V$ that contain r, of the number of edges leaving S.*

The following theorem due to Lovász [396] states an interrelation of the maximum number of directed edge-disjoint paths and the in- and out-degree of a vertex.

Theorem 7.1.14 (Lovász, 1973). *Let $v \in V$ be a vertex of a graph $G = (V, E)$. If $\lambda_G(v, w) \leq \lambda_G(w, v)$ for all vertices $w \in V$, then $d^+(v) \leq d^-(v)$.*

As an immediate consequence, this theorem provided a proof for Kotzig's conjecture:

Theorem 7.1.15 (Kotzig's Theorem). *For a directed graph G, $\lambda_G(v, w)$ equals $\lambda_G(w, v)$ for all $v, w \in V$ if and only if the graph is pseudo-symmetric, i.e. the in-degree equals the out-degree for all vertices: $d^+(v) = d^-(v)$.*

7.2 Introduction to Minimum Cuts

For short, in an undirected weighted graph the sum of the weights of the edges with one endpoint in each of two disjoint vertex sets X and Y is denoted by $w(X, Y)$. For directed graphs, $w(X, Y)$ is defined in nearly the same way, but we only count the weight of edges with their origin in X and their destination in Y. A cut in a weighted graph $G = (V, E)$ is a set of vertices $\emptyset \subset S \subset V$ and its weight is $w(S, V \setminus S)$. In an unweighted graph, the weight of a cut is the number of edges from S to $V \setminus S$.

Definition 7.2.1. *A* minimum cut *is a cut S such that for all other cuts T,*

$$w(S, V \setminus S) \leq w(T, V \setminus T).$$

Observation 7.2.2. *A minimum cut in a connected graph G with edge weights greater than zero induces a connected subgraph of G.*

An algorithm that computes all *minimum cuts* has to represent these cuts. A problem is to store all minimum cuts without using too much space. A suggestion was made in 1976 by Dinitz et al. [153]. They presented a data structure called *cactus* that represents all minimum cuts of an undirected (weighted) graph. The size of a cactus is linear in the number of vertices of the input graph and a cactus allows us to compute a cut in a time linear in the size of the cut.

Karzanov and Timofeev outlined in [351] a first algorithm to construct a cactus for unweighted, undirected graphs. Their algorithm consists of two parts. Given an arbitrary input graph G, the first part finds a sequence of all minimum cuts in G and the second constructs the cactus C_G from this sequence. The algorithm also works on weighted graphs, as long as all weights are positive.

If negative weights are allowed, the problem of finding a minimum cut is $\mathcal{NP}$-hard [345]. Moreover, no generalization for directed graphs is known. An unweighted graph can be reduced to a weighted graph by assigning weight 1 to all edges. In the following, we will therefore consider the problem of finding minimum cuts only for undirected connected graphs with positive weights.

Consider a network N defined by the directed graph $G = (V, E)$, a capacity function u_N, a source s, a sink t and a flow f (Chapter 2). A *residual network* R_f consists of those edges that can carry additional flow, beyond what they already carry under f. Thus R_f is defined on the graph $G_{R_f} := \left(V, \left\{(u,v) \,\middle|\, ((u,v) \in E \vee (v,u) \in E) \wedge u_{R_f}((u,v)) > 0\right\}\right)$ with the same source s and sink t and the following capacity function

$$u_{R_f}((a,b)) := \begin{cases} c(a,b) - f(a,b) + f(b,a) & \text{if } (a,b) \in E \wedge (b,a) \in E \\ c(a,b) - f(a,b) & \text{if } (a,b) \in E \wedge (b,a) \notin E \\ f(b,a) & \text{if } (a,b) \notin E \wedge (b,a) \in E \end{cases}$$

Let $R_{f_{max}}$ be the residual network of N and f_{max}, where f_{max} is a maximum s-t-flow in N. As a consequence of Theorem 2.2.1 on page 11, the maximum flow saturates all minimum s-t-cuts and therefore each set $S \subseteq V \setminus t$ is a minimum s-t-cut iff $s \in S$ and no edges leave S in $R_{f_{max}}$.

7.3 All-Pairs Minimum Cuts

The problem of computing a minimum cut between all pairs of vertices can, of course, easily be done by solving $n(n-1)/2$ flow problems. As has been shown by Gomory and Hu [257], the computation of $n-1$ maximum flow problems is already sufficient to determine the value of a maximum flow / minimum cut for all pairs of vertices. The result can be represented in the *equivalent flow tree*, which is a weighted tree on n vertices, where the minimum weight of any edge on the (unique) path between two vertices s and t equals the maximum flow from s to t. They furthermore showed that there always exists an equivalent flow tree,

where the components that result from removing the minimum weight edge of the s-t-path represent a minimum cut between s and t. This tree is called the *Gomory-Hu cut tree*.

Gusfield [265] demonstrated how to do the same computation without node contractions and without the overhead for avoiding the so called crossing cuts. See also [272, 344, 253].

If one is only interested in any edge cutset of minimum weight in an undirected weighted graph (without a specified vertex pair to be disconnected), this can be done using the algorithm of Stoer and Wagner, see Section 7.7.1.

7.4 Properties of Minimum Cuts in Undirected Graphs

There are $2^{|V|}$ sets and each of them is possibly a minimum cut, but the number of minimum cuts in a fixed undirected graph is polynomial in $|V|$. To see this, we need to discuss some well-known facts about minimum cuts. These facts also help us to define a data structure called *cactus*. A cactus can represent all minimum cuts, but needs only space linear in $|V|$.

For short, for a graph G, let in this chapter λ_G always denote the weight of a minimum cut. If the considered graph G is clear from the context, the index G of λ_G is omitted.

Lemma 7.4.1. *Let S be a minimum cut in $G = (V, E)$. Then, for all $\emptyset \neq T \subset S : w(T, S \setminus T) \geq \frac{\lambda}{2}$.*

Proof. Assume $w(T, S \setminus T) < \frac{\lambda}{2}$. Since $w(T, V \setminus S) + w(S \setminus T, V \setminus S) = \lambda$, w.l.o.g. $w(T, V \setminus S) \leq \frac{\lambda}{2}$ (if not, define T as $S \setminus T$). Then $w(T, V \setminus T) = w(T, S \setminus T) + w(T, V \setminus S) < \lambda$. Contradiction. □

Lemma 7.4.2. *Let $A \neq B$ be two minimum cuts such that $T := A \cup B$ is also a minimum cut. Then*

$$w\left(A, \bar{T}\right) = w\left(B, \bar{T}\right) = w(A \setminus B, B) = w(A, B \setminus A) = \frac{\lambda}{2}.$$

Proof. As in the Figure 7.2, let $a = w\left(A, \bar{T}\right)$, $b = w\left(B, \bar{T}\right)$, $\alpha = w(A, B \setminus A)$ and $\beta = w(B, A \setminus B)$. Then $w\left(A, \bar{A}\right) = a + \alpha = \lambda$, $w\left(B, \bar{B}\right) = b + \beta = \lambda$ and $w\left(T, \bar{T}\right) = a + b = \lambda$. We also know that $w\left(A \setminus B, B \cup \bar{T}\right) = a + \beta \geq \lambda$ and $w\left(B \setminus A, A \cup \bar{T}\right) = b + \alpha \geq \lambda$. This system of equations and inequalities has only one unique solution: $a = \alpha = b = \beta = \frac{\lambda}{2}$. □

Definition 7.4.3. *A pair $\langle S_1, S_2 \rangle$ is called crossing cut, if S_1, S_2 are two minimum cuts and neither $S_1 \cap S_2$, $S_1 \setminus S_2$, $S_2 \setminus S_1$ nor $\bar{S_1} \cap \bar{S_2}$ is empty.*

Lemma 7.4.4. *Let $\langle S_1, S_2 \rangle$ be crossing cuts and let $A = S_1 \cap S_2$, $B = S_1 \setminus S_2$, $C = S_2 \setminus S_1$ and $D = \bar{S_1} \cap \bar{S_2}$. Then*

a. A, B, C and D are minimum cuts

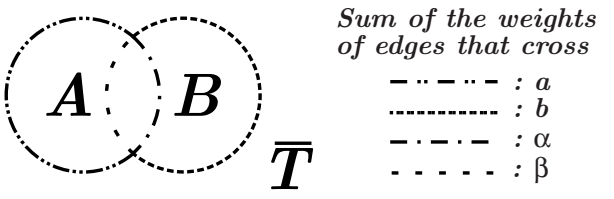


Fig. 7.2. Intersection of two minimum cuts A and B

b. $w(A, D) = w(B, C) = 0$
c. $w(A, B) = w(B, D) = w(D, C) = w(C, A) = \frac{\lambda}{2}$.

Proof. Since we know that S_1 and S_2 are minimum cuts, we can conclude

$$w\left(S_1, \bar{S_1}\right) = w(A, C) + w(A, D) + w(B, C) + w(B, D) = \lambda$$
$$w\left(S_2, \bar{S_2}\right) = w(A, B) + w(A, D) + w(B, C) + w(C, D) = \lambda$$

and since there is no cut with weight smaller than λ, we know that

$$w\left(A, \bar{A}\right) = w(A, B) + w(A, C) + w(A, D) \geq \lambda$$
$$w\left(B, \bar{B}\right) = w(A, B) + w(B, C) + w(B, D) \geq \lambda$$
$$w\left(C, \bar{C}\right) = w(A, C) + w(B, C) + w(C, D) \geq \lambda$$
$$w\left(D, \bar{D}\right) = w(A, D) + w(B, D) + w(C, D) \geq \lambda$$

Summing up twice the middle and the right side of the first two equalities we obtain

$$2 \cdot w(A, B) + 2 \cdot w(A, C) + 4 \cdot w(A, D) + 4 \cdot w(B, C) + 2 \cdot w(B, D) + 2 \cdot w(C, D) = 4 \cdot \lambda$$

and summing up both side of the four inequalities we have

$$2 \cdot w(A, B) + 2 \cdot w(A, C) + 2 \cdot w(A, D) + 2 \cdot w(B, C) + 2 \cdot w(B, D) + 2 \cdot w(C, D) \geq 4 \cdot \lambda$$

Therefore $w(A, D) = w(B, C) = 0$. In other words, there are no diagonal edges in Figure 7.3.

For a better imagination, let us assume that the length of the four inner line segments in the figure separating A, B, C and D is proportional to the sum of the weights of all edges crossing this corresponding line segments. Thus the total length l of both horizontal or both vertical lines, respectively, is proportional to the weight λ.

Let us assume the four line segments have different length, in other words, the two lines separating the sets S_1 from $\bar{S_1}$ or S_2 from $\bar{S_2}$, respectively, do not cross each other exactly in the midpoint of the square, then the total length of the separating line segments of one vertex set $\Delta = A, B, C$ or D is shorter then l. Thus $w(\Delta, \bar{\Delta}) < \lambda$. Contradiction.

As a consequence, $w(A, B) = w(B, D) = w(D, C) = w(C, A) = \frac{\lambda}{2}$ and $w\left(A, \bar{A}\right) = w\left(B, \bar{B}\right) = w\left(C, \bar{C}\right) = w\left(D, \bar{D}\right) = \lambda$. □

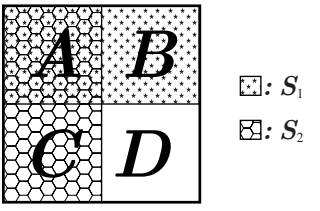


Fig. 7.3. Crossing cuts $\langle S_1, S_2 \rangle$ with $S_1 := A \cup B$ and $S_2 := A \cup C$

A crossing cut in $G = (V, E)$ partitions the vertex set V into exactly four parts. A more general definition is the following, where the vertex set can be divided in three or more parts.

Definition 7.4.5. *A* circular partition *is a partition of V into $k \geq 3$ disjoint sets $V_1, V_2, \ldots, V_k$ such that*

a. $w(V_i, V_j) = \begin{cases} \lambda/2 & : \quad |i-j| = 1 \bmod k \\ 0 & : \quad otherwise \end{cases}$

b. If S is a minimum cut, then

1. *S or $\bar{S}$ is a proper subset of some V_i or*
2. *the circular partition is a refinement of the partition defined by the minimum cut S. In other words, the minimum cut is the union of some of the sets of the circular partition.*

Let $V_1, V_2, \ldots, V_k$ be the disjoint sets of a *circular partition*, then for all $1 \leq a \leq b < k, S := \left(\cup_{i=a}^{b} V_i\right)$ is a minimum cut. Of course, the complement of S containing V_k is a minimum cut, too. Let us define these minimum cuts as *circular partition cuts*. Especially each V_i, $1 \leq i \leq k$, is a minimum cut (property a. of the last definition).

Consider a minimum cut S such that neither S nor its complement is contained in a set of the circular partition. Since S is connected (Observation 7.2.2), S or its complement are equal to $\cup_{i=a}^{b} V_i$ for some $1 \leq a < b < k$.

Moreover, for all sets V_i of a circular partition, there exists no minimum cut S such that $\langle V_i, S \rangle$ is a crossing cut (property b. of the last definition).

Definition 7.4.6. *Two different circular partitions $P := \{U_1, \ldots, U_k\}$ and $Q := \{V_1, \ldots, V_l\}$ are* compatible *if there is a unique r and s, $1 \leq r, s \leq k$, such that for all $i \neq r : U_i \subseteq V_s$ and for all $j \neq s : V_j \subseteq U_r$.*

Lemma 7.4.7 ([216]). *All different circular partitions are pairwise compatible.*

Proof. Consider two circular partitions P and Q in a graph $G = (V, E)$. All sets of the partitions are minimum cuts. Assume a set $S \in P$ is equal to the union of more than one and less than all sets of Q. Exactly two sets $A, B \in Q$ contained in S are connected by at least an edge to the vertices $V \setminus S$. Obtain T from S by replacing $A \subset S$ by an element of Q connected to B and not contained in S. Then $\langle S, T \rangle$ is a crossing cut, contradiction.

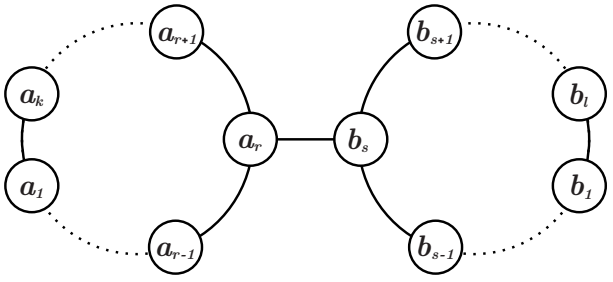


Fig. 7.4. Example graph $G = (\{a_1 \dots a_r, b_1 \dots b_s\}, E)$ shows two compatible partitions P, Q defined as follows:

$$P := \{\{a_1\}, \dots, \{a_{r-1}\}, \{a_r, b_1, \dots b_l\}, \{a_{r+1}\}, \dots \{a_k\}\}$$
$$Q := \{\{b_1\}, \dots, \{b_{s-1}\}, \{b_s, a_1, \dots a_k\}, \{b_{s+1}\}, \dots \{b_l\}\}$$

Therefore each set of P or its complement is contained in some set of Q.

Assume two sets of P are contained in two different sets of Q. Since each complement of the remaining sets of P cannot be contained in one set of Q, each remaining set of P must be contained in one subset of Q. Thus, $P = Q$. Contradiction.

Assume now all sets of P are contained in one set Y of Q. Then $Y = V$. Again a contradiction.

Since the union of two complements of sets in P is V and Q contains at least three sets, only one complement can be contained in one set of Q. Thus, there is exactly one set X of P that is not contained in Y of Q, but $\bar{X} \subset Y$. □

Lemma 7.4.8. *If S_1, S_2 and S_3 are pairwise crossing cuts, then*

$$S_1 \cap S_2 \cap S_3 = \emptyset.$$

Proof. Assume that the lemma is not true. As shown in Figure 7.5, let

$$a = w\left(S_3 \setminus (S_1 \cup S_2), \overline{S_1 \cap S_2 \cap S_3}\right)$$
$$b = w\left((S_2 \cap S_3) \setminus S_1, S_2 \setminus (S_1 \cup S_3)\right)$$
$$c = w\left(S_1 \cap S_2 \cap S_3, (S_1 \cap S_2) \setminus S_3\right)$$
$$d = w\left((S_1 \cap S_3) \setminus S_2, S_1 \setminus (S_2 \cup S_3)\right)$$

On one hand $S_1 \cap S_2$ is a minimum cut (Lemma 7.4.4.a.) so that $c \geq \frac{\lambda}{2}$ (Lemma 7.4.1). On the other hand $c+b = c+d = \frac{\lambda}{2}$ (Lemma 7.4.4.c.). Therefore $b = d = 0$ and $(S_1 \cap S_3) \setminus S_2 = (S_2 \cap S_3) \setminus S_1 = \emptyset$.

If we apply Lemma 7.4.4.b. to S_1 and S_2, then $S_1 \cap S_2 \cap S_3$ and $S_3 \setminus (S_1 \cup S_2)$ are not connected. Contradiction. □

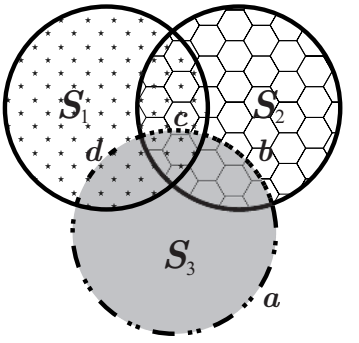


Fig. 7.5. Three pairwise crossing cuts S_1,S_2 and S_3

Lemma 7.4.9. *If S_1, S_2 and T are minimum cuts with $S_1 \subset S_2$, $T \not\subset S_2$ and $\langle S_1, T\rangle$ is a crossing cut, then $A := (S_2 \setminus S_1) \setminus T$, $B := S_1 \setminus T$, $C := S_1 \cap T$ and $D := (S_2 \setminus S_1) \cap T$ are minimum cuts, $w(A,B) = w(B,C) = w(C,D) = \frac{\lambda}{2}$ and $w(A,C) = w(A,D) = w(B,D) = 0$.*

Proof. Since $\langle S_1, T\rangle$ and therefore $\langle S_2, T\rangle$ is a crossing cut,

$$w(A \cup B, C \cup D) = \frac{\lambda}{2}\ (1),\ w(B,C) = \frac{\lambda}{2}\ (2),$$

$$w\left(A,B\right) + w\left(B, \overline{S_1 \cup S_2}\right) = w\left(B, A \cup \overline{S_1 \cup S_2}\right) = \frac{\lambda}{2}\ (3) \text{ and}$$

$$w\left(A, \overline{S_1 \cup S_2}\right) + w\left(B, \overline{S_1 \cup S_2}\right) = w\left(A \cup B, \overline{S_1 \cup S_2}\right) = \frac{\lambda}{2}\ (4).$$

All equalities follow from Lemma 7.4.4.c.. Moreover $w\left(A, T \setminus S_2\right) = 0$, $w\left(D, \overline{S_1 \cup S_2}\right) = 0$ (7.4.4.b.) and B, C are minimum cuts. Since (1), (2) and

$$w(A \cup B, C \cup D) = w(A,C) + w(A,D) + w(B,C) + w(B,D),$$

we can conclude that $w(A,C) = w(A,D) = w(B,D) = 0$.

A consequence of (3) and (4) is $w\left(A, \overline{S_1 \cup S_2}\right) = w\left(A,B\right)$. Moreover, $w\left(A,B\right) \geq \frac{\lambda}{2}$ (Lemma 7.4.1) and $w\left(A, \overline{S_1 \cup S_2}\right) \leq w\left(A, \overline{S_1} \cup S_2\right) = \frac{\lambda}{2}$. Therefore $w\left(A, \overline{S_1 \cup S_2}\right) = w(A,B) = \frac{\lambda}{2}$ and A is a minimum cut.

With a similar argument we can see, $w(C,D) = \frac{\lambda}{2}$ and D is a minimum cut. Therefore, the general case shown in Figure 7.6(a) can always be transformed into the Figure 7.6(b). □

For short, given some sets $S_1, \ldots, S_k$, let

$$\mathcal{F}^{\alpha_1,\ldots,\alpha_k}_{S_1,\ldots S_k} = \bigcap_{i=1}^{k} \left\{ \begin{matrix} S_i \text{ if } \alpha_i = 1 \\ \overline{S_i} \text{ if } \alpha_i = 0 \end{matrix} \right\} \text{ and}$$

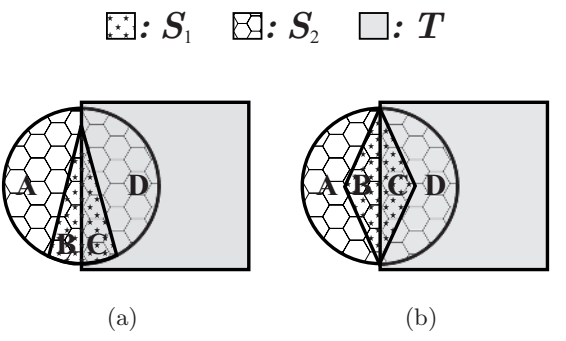


Fig. 7.6. Intersection of three minimum cuts

$$\mathcal{F}_{\{S_1,\dots,S_k\}} = \left(\bigcup_{\alpha_1,\dots,\alpha_k \in \{0,1\}^k} \mathcal{F}^{\alpha_1,\dots,\alpha_k}_{\{S_1,\dots,S_k\}} \right) \setminus \{\emptyset\}.$$

Lemma 7.4.10. *Let $\langle S_1, S_2\rangle$ be a crossing cut and $A \in \mathcal{F}_{\{S_1,S_2\}}$. Choose $B \in \mathcal{F}_{\{S_1,S_2\}}$ such that $w(A,B) = \frac{\lambda}{2}$. For all crossing cuts $\langle B, T\rangle$:*

$$w(A, B \cap T) = \frac{\lambda}{2} \text{ or } w\left(A, B \cap \bar{T}\right) = \frac{\lambda}{2}$$

Proof. W.l.o.g. $A = S_1 \cap S_2$ (if not, interchange S_1 and $\bar{S_1}$ or S_2 and $\bar{S_2}$), $B = S_1 \setminus S_2$ (if not, interchange S_1 and S_2). Let $C = S_2 \setminus S_1$ and $D = \bar{S_1} \cap \bar{S_2}$. Then $(*) : w(B,C) = 0$ (Lemma 7.4.4.b.). Consider the following four cases:

$\mathbf{T \subset (A \cup B)}$ (Figure 7.7(a)) **:** $w(A, B \cap T) = \frac{\lambda}{2}$ (Lemma 7.4.9)

$\mathbf{T \cap D \neq \emptyset}$ **:** Because $\langle S_1, T\rangle$ is a crossing cut,

$$\begin{aligned} &w(A \setminus T, A \cap T) + w(A \setminus T, B \cap T) + w(B \setminus T, A \cap T) + w(B \setminus T, B \cap T) \\ &= w((A \setminus T) \cup (B \setminus T), (A \cap T) \cup (B \cap T)) \\ &= w(S_1 \setminus T, S_1 \cap T) = \frac{\lambda}{2}. \end{aligned}$$

Together with $w(B \setminus T, B \cap T) \geq \frac{\lambda}{2}$ (Lemma 7.4.1), we can conclude
- $w(A \setminus T, A \cap T) = 0$ and therefore $A \cap T = \emptyset$ or $A \setminus T = \emptyset$,
- $w(A \setminus T, B \cap T) = 0$ (1) and
- $w(A \cap T, B \setminus T) = 0$ (2).

Note that $w(A,B) = \frac{\lambda}{2}$. If $A \cap T = \emptyset$, $w(A, B \cap T) \overset{(1)}{=} 0$ and $w(A, B \setminus T) = \frac{\lambda}{2}$. Otherwise $A \setminus T = \emptyset$, $w(A, B \setminus T) \overset{(2)}{=} 0$ and $w(A, B \cap T) = \frac{\lambda}{2}$.

$\mathbf{T} \not\subset (\mathbf{A} \cup \mathbf{B})$ **and** $\mathbf{T} \cap \mathbf{D} = \emptyset$ (3) **and** $(\mathbf{A} \cup \mathbf{C}) \subset \mathbf{T}$ (4) (Figure 7.7(b)) **:**

$$w(A, T \cap B) \overset{(*)}{=} w(A \cup C, T \cap B) \overset{(3),(4)}{=} w((A \cup C) \cap T, T \setminus (A \cup C)) \geq \frac{\lambda}{2},$$

since $(A \cup C)$ is a minimum cut (Lemma 7.4.1). Using the fact $w(A, B) = \frac{\lambda}{2}$, we get $w(A, T \cap B) = \frac{\lambda}{2}$.

$\mathbf{T} \not\subset (\mathbf{A} \cup \mathbf{B})$ **and** $\mathbf{T} \cap \mathbf{D} = \emptyset$ (5) **and** $(\mathbf{A} \cup \mathbf{C}) \not\subset \mathbf{T}$ (Figure 7.7(c)) **:**

$$w(A, T \cap B) \overset{(*)}{=} w(A \cup C, T \cap B) \overset{(5)}{=} w(A \cup C, T \setminus (A \cup C)) = \frac{\lambda}{2},$$

since $\langle A \cup C, T \rangle$ is a crossing cut.

This concludes the proof. □

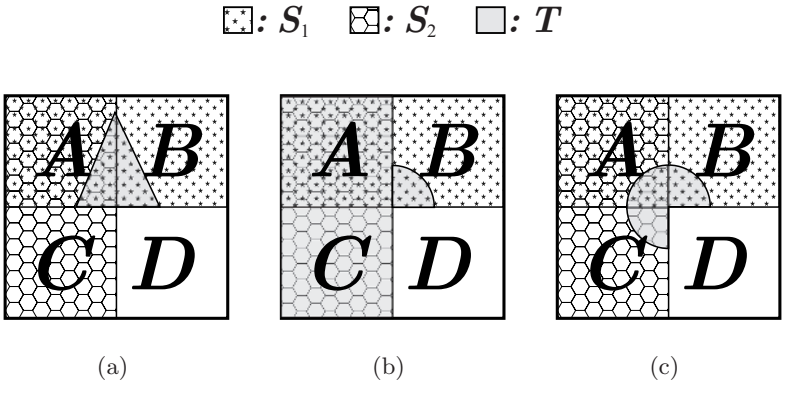


Fig. 7.7. A minimum cut T and a crossing cut $\langle S_1, S_2 \rangle$

Corollary 7.4.11. *The intersection of a crossing cut partitions the vertices of the input graph into four minimum cuts. Lemma 7.4.4.c. guarantees us that for each of the four minimum cuts A there exist two of the three remaining minimum cuts B, C such that* $w(A, B) = w(A, C) = \frac{\lambda}{2}$. *Although set B or C may be divided in smaller parts by further crossing cuts, there are always exactly two disjoint minimum cuts* $X \subseteq B$ *and* $Y \subseteq C$ *with* $w(A, X) = w(A, Y) = \frac{\lambda}{2}$.

Proof. Assume the corollary is not true. Let $\langle S, X_{1\&2} \rangle$ be the first crossing cut that divides the set $X_{1\&2}$ with $w(A, X_{1\&2}) = \frac{\lambda}{2}$ into the two disjoint sets X_1, X_2 with $w(A, X_1), w(A, X_2) \geq 0$. But then $\langle S, B \rangle$ or $\langle \bar{S}, B \rangle$ is also a crossing cut, which divides B into B_1 and B_2 with $X_1 \subseteq B_1$ and $X_2 \subseteq B_2$. Thus, $w(A, B_1), w(A, B_2) \geq 0$. This is a contradiction to Lemma 7.4.10. □

Different crossing cuts interact in a very specific way, as shown in the next theorem.

Theorem 7.4.12 ([63, 153]). *In a graph $G = (V, E)$, for each partition P of V into 4 disjoint sets due to a crossing cut in G, there exists a circular partition in G that is a refinement of P.*

Proof. Given crossing cut $\langle S_1, S_2 \rangle$, choose the set

$$\Lambda := \left\{ S_1 \cap S_2, S_1 \setminus S_2, S_2 \setminus S_1, \overline{S_1 \cup S_2} \right\}$$

as a starting point.

As long as there is a crossing cut $\langle S, T \rangle$ for some $T \notin \Lambda$ and $S \in \Lambda$, add T to Λ. This process terminates since we can only add each set $T \in \mathcal{P}(V)$ into Λ once. All sets in Λ are minimum cuts. Definition 7.4.5.b. is satisfied for Λ.

The disjoint minimum cuts $\mathcal{F}(\Lambda)$ give us a partitioning of the graph. All sets in $\mathcal{F}(\Lambda)$ can be built by crossing cuts of minimum cuts in Λ. Therefore, each set in $\mathcal{F}(\Lambda)$ has exactly two neighbors, i.e., for each set $X \in \mathcal{F}(\Lambda)$, there exist exactly two different sets $Y, Z \in \mathcal{F}(\Lambda)$ such that $w(X, Y) = w(X, Z) = \frac{\lambda}{2}$ (Corollary 7.4.11). For all other sets $Z \in \mathcal{F}(\Lambda)$, $w(X, Z) = 0$. Since G is a connected graph, all sets in $\mathcal{F}(\Lambda)$ can be ordered, so that Definition 7.4.5.a. holds. Observe that Definition 7.4.5.b. is still true, since splitting the sets in Λ into smaller sets still allows a reconstruction of the sets in Λ. □

Lemma 7.4.13 ([63, 153]). *A graph $G = (V, E)$ has $\mathcal{O}\left(\binom{|V|}{2}\right)$ many minimum cuts and this bound is tight. This means that a graph can have $\Omega\left(\binom{|V|}{2}\right)$ many minimum cuts.*

Proof. The upper bound is a consequence of the last theorem. Given a graph $G = (V, E)$, the following recursive function Z describes the number of minimum cuts in G:

$$Z(|V|) = \begin{cases} \sum_{i=1}^{k} \left(Z(|V_i|)\right) + \binom{k}{2} & \text{A circular partition } V_1, \ldots, V_k \text{ exists in } G \\ Z(|S|) + Z(|V - S|) + 1 & \text{No circular partition, but a minimum cut } S \text{ exists in } G \\ 0 & \text{otherwise} \end{cases}$$

It is easy to see that this function achieves the maximum in the case where a circular partition $W_1, \ldots, W_{|V|}$ exist. Therefore $Z(|V|) = \mathcal{O}\left(\binom{|V|}{2}\right)$.

The lower bound is achieved by a simple cycle of n vertices. There are $\Omega\left(\binom{n}{2}\right)$ pairs of edges. Each pair of edges defines another two minimum cuts S and $\overline{S}$. These two sets are separated by simply removing the pair of edges. □

7.5 Cactus Representation of All Minimum Cuts

In the following, a description of the *cactus* is given. First consider a graph $G = (V, E)$ without any circular partitions. Then due to the absence of all crossing cuts, all minimum cuts of G are laminar.

A set $\mathcal{S}$ of sets is called *laminar* if for every pair of sets $S_1, S_2 \in \mathcal{S}$, either S_1 and S_2 are disjoint or S_1 is contained in S_2 or vice versa. Therefore each set $T \in \mathcal{S}$ contained in some $S_1, S_2, \ldots \in \mathcal{S}$ has a unique smallest superset. For clarity, we say that a tree has nodes and leaves, while a graph has vertices. Each laminar set $\mathcal{S}$ can be represented in a tree. Each node represents a set in $\mathcal{S}$; the leaves represent the sets in $\mathcal{S}$ that contain no other sets of $\mathcal{S}$. The parent of a node representing a set T represents the smallest superset of T. This construction ends with a set of trees called forest. Add an extra node r to the forest and connect all roots of the trees of the forest by an edge to this new node r, which is now the root of one big tree. Therefore, the nodes of one tree represent all sets of $\mathcal{S}$, and the root of the tree represents the entire underlying set, i.e. the union of all elements of all $S \in \mathcal{S}$. If this union has n elements, then such a tree can have at most n leaves and therefore at most $2n - 1$ nodes.

Since all minimum cuts G are laminar, these can be represented by a tree T_G defined as follows. Consider the smaller vertex set of every minimum cut. Denote this set of sets as Λ. If the vertex sets of a minimum cut are of same size, take one of these sets. Represent each set of Λ by a single node. Two nodes corresponding to minimum cuts A and B in G are connected by an edge if $A \subset B$ and there is no other minimum cut C such that $A \subset C \subset B$. The roots of the forest represent the minimum cuts in Λ that are contained in no other minimum cut in Λ. Again, connect all roots of the forest by an edge to a single extra node that we define as root of the tree.

Because removing one edge in the tree separates a subtree from the rest of the tree, let us define the following mapping: each vertex of the graph G is mapped to the node of the tree T_G that corresponds to the smallest cut containing this vertex. All vertices that are contained in no node of T_G are mapped to the root of T_G.

For each minimum cut S of G, the vertices of S are then mapped to some set of nodes X such that there is an edge and removing this edge separates the nodes X from the rest of the tree. Conversely, removing one edge from T_G separates the nodes of the tree into two parts such that the set of all vertices mapped into one part is a minimum cut.

If G has no circular partitions, the tree T_G is the *cactus* C_G for G. The number of nodes of a cactus is bounded by $2|V| - 1$.

Consider a graph $G = (V, E)$ that has only one circular partition $V_1, \ldots V_k$. The circular partition cuts can be represented by a circle of k nodes. For $1 \leq i \leq k$, the vertices of each part V_i are represented by one node N_i of the circle in such a way that two parts V_i and V_{i+1} are represented by two adjacent nodes.

Now we make use of the fact that for each minimum cut S that is no circular partition cut, either S or $\bar{S}$ is a proper subset of a V_i. Therefore, we can construct the tree $T_{(V_i,E)}$ for all minimum cuts that are a subset of V_i, but now with the

restriction that only the vertices of V_i are mapped to this tree. The root of $T_{(V_i,E)}$ corresponds exactly to the set V_i. Thus we can merge node N_i of the circle and the root of $T_{(V_i,E)}$ for all $1 \leq i \leq k$. This circle connected with all the trees is the cactus C_G for G. The number of nodes is equal to the sum of all nodes in the trees $T_{(V_i,E)}$ with $1 \leq i \leq k$. Therefore, the number of nodes of the cactus is bounded by $2|V| - 1$ and again, there is a $1 - 1$ correspondence between minimum cuts in G and the separation of C_G into two parts.

Now consider a graph $G = (V, E)$ with the circular partitions $P_1, \ldots, P_z$. Take all circular partitions as a set of sets. Construct a cactus C_G representing the circular partition cuts of G in the following way.

The vertices of each set $F \in \mathcal{F}_{P_1 \cup \ldots \cup P_z}$ are mapped to one node and two nodes are connected, if for their corresponding sets F_1 and F_2, $w(F_1, F_2) > 0$. Then each circular partition creates one circle in C_G. Since all circular partitions are pairwise compatible, the circles are connected by edges that are not part of any circle. The cactus C_G is now a tree-like graph (Figure 7.8).

After representing the remaining minimum cuts that are not part of a circular partition, we get the cactus T_C for G. As before, the number of nodes of the cactus is bounded by $2|V| - 1$.

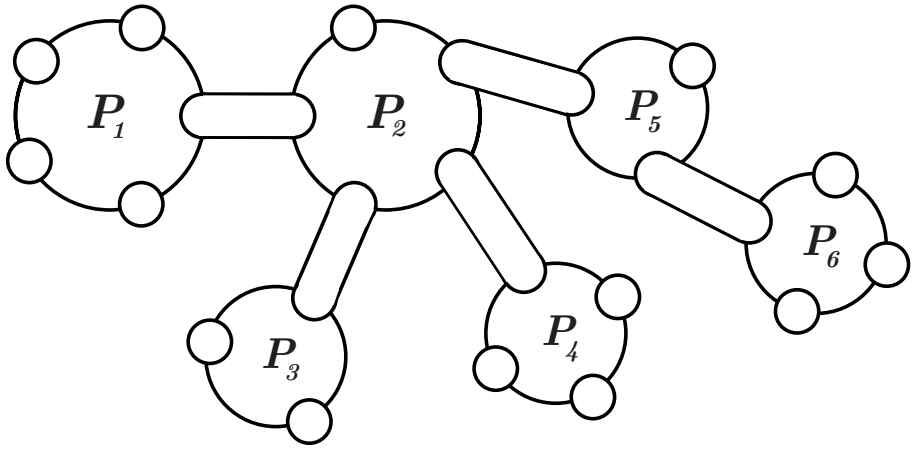


Fig. 7.8. A cactus representing the circular partition cuts of 6 circular partitions

7.6 Flow-Based Connectivity Algorithms

We distinguish algorithms that check k-vertex/edge-connectivity of a graph G for a given natural number k, and algorithms that compute the vertex/edge-connectivity $\kappa(G)$ or $\lambda(G)$ respectively. (A third kind of algorithms computes the maximal k-vertex/edge-connected subgraphs (k-components), which is the subject of discussion in Section 7.8.)

Most of the algorithms for computing vertex- or edge-connectivities are based on the computation of the maximum flow through a derived network. While the flow problem in undirected graphs can be reduced to a directed flow problem of comparable size [220], for the other direction only a reduction with increased capacities is known [478]. There were several algorithms published for the solution of (general) flow problems, see Table 7.1.

Table 7.1. The history of max-flow algorithms

1955 Dantzig & Fulkerson		[231, 141]
Network simplex method	$\mathcal{O}(n^2mU)$	[140, 139]
1956 Ford & Fulkerson		[218, 219]
Augmenting path / Labeling	$\mathcal{O}(nmU)$	[220]
1969 Edmonds & Karp		[172]
Shortest augmenting path	$\mathcal{O}(nm^2)$	[593]
Capacity scaling	$\mathcal{O}(m^2 \log U)$	
1970 Dinitz		[150]
Layered network / blocking flow	$\mathcal{O}(n^2m)$	
1973 Dinitz		[151, 234]
Capacity scaling	$\mathcal{O}(nm \log U)$	
1974 Karzanov		[350]
Preflow-push / layered network	$\mathcal{O}(n^3)$	
1977 Cherkassky	$\mathcal{O}(n^2\sqrt{m})$	[122, 123]
1978 Malhotra, Kumar, Maheshwari	$\mathcal{O}(n^3)$	[406]
1978 Galil	$\mathcal{O}(n^{5/3}m^{2/3})$	[236]
1979 Galil & Naamad / Shiloach	$\mathcal{O}(nm(\log n)^2)$	[238, 518]
1980 Sleater & Tarjan		[525]
Dynamic trees	$\mathcal{O}(nm \log n)$	
1985 Goldberg		[249]
Push-relabel	$\mathcal{O}(n^3)$	
1986 Goldberg & Tarjan		[252]
Push-relabel	$\mathcal{O}(nm \log(n^2/m))$	
1987 Ahuja & Orlin		[7]
Excess scaling	$\mathcal{O}(nm + n^2 \log U)$	
1990 Cheriyan, Hagerup, Mehlhorn		[119]
Incremental algorithm	$\mathcal{O}(n^3/\log n)$	
1990 Alon		[118, 20]
Derandomization	$\mathcal{O}(nm + n^{8/3} \log n)$	
1992 King, Rao, Tarjan		[118, 356]
Online game	$\mathcal{O}(nm + n^{2+\epsilon})$	
1993 Phillips & Westbrook		[476]
Online game	$\mathcal{O}(nm \log_{m/n} n + n^2 \log^{2+\varepsilon} n)$	
1998 Goldberg & Rao		[250]
Non-unit length function	$\mathcal{O}(\min(n^{2/3}, \sqrt{m})m \log \frac{n^2}{m} \log U)$	

U denotes the largest possible capacity (integer capacities case only)

Better algorithms for the more restricted version of unit capacity networks exist.

Definition 7.6.1. *A network is said to be a* unit capacity network *(or* 0-1 network*) if the capacity is* 1 *for all edges. A unit capacity network is of* type 1 *if it has no parallel edges. It is called* type 2 *if for each vertex* v *(*$v \neq s$, $v \neq t$*) either the in-degree* $d^-(v)$ *or the out-degree* $d^+(v)$ *is only* 1*.*

Lemma 7.6.2. *1. For unit capacity networks, the computation of the maximum flow can be done (using Dinitz's algorithm) in* $\mathcal{O}(m^{3/2})$*.*
2. For unit capacity networks of type 1, the time complexity of Dinitz's algorithm is $\mathcal{O}(n^{2/3}m)$*.*
3. For unit capacity networks of type 2, the time complexity of Dinitz's algorithm is $\mathcal{O}(n^{1/2}m)$*.*

For a proof of the lemma see [188, 187, 349].

While the best bound for directed unit capacity flow problems differs only by logarithmic factors from the best known bound for integer capacities, even better bounds for the case of undirected unit capacity networks exist: $\mathcal{O}(\min(m, n^{3/2})\sqrt{m})$ by Goldberg and Rao [251], $\mathcal{O}(n^{7/6}m^{2/3})$ by Karger and Levine [343].

7.6.1 Vertex-Connectivity Algorithms

Table 7.2. The history of computing the vertex-connectivity κ

Year	Author(s)	MaxFlow calls	Compute κ	Ref.
1974	Even & Tarjan	$(\kappa+1)(n-\delta-1)$	$\mathcal{O}(\kappa n^{3/2}m)$ $\mathcal{O}(n^{1/2}m^2)$	[188]
1984	Esfahanian & Hakimi	$n-\delta-1+\kappa(2\delta-\kappa-3)/2$	$\mathcal{O}((n-\delta+\kappa\delta-\kappa^2/2)\cdot n^{2/3}m)$	[183]
1996	Henzinger, Rao, Gabow		$\mathcal{O}(\min\{\kappa^3+n, \kappa n\}\kappa n)$	[298]

Table 7.3. The history of checking vertex-connectivity

Year	Author(s)	MaxFlow calls	Check k-VC	Ref.
1969	Kleitman	$k(n-\delta)-\binom{k+1}{2}$	$\mathcal{O}(k^2n^3)$	[362]
1973	Even	$n-k+\binom{k}{2}$	$\mathcal{O}(k^3m+knm)$	[186]
1984	Esfahanian & Hakimi	$n-k+\binom{k-1}{2}$	$\mathcal{O}(k^3m+knm)$	[183]

The basis of all flow-based connectivity algorithms is a subroutine that computes the local connectivity between two distinct vertices s and t. Even [185, 186,

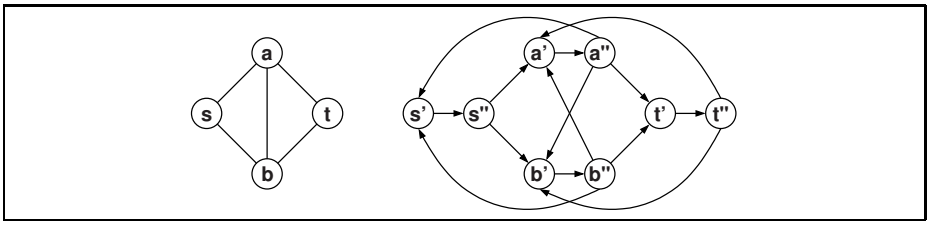


Fig. 7.9. Construction of the directed graph $\bar{G}$ that is derived from the undirected input graph G to compute the local vertex-connectivity $\kappa_G(s,t)$

187] presented a method for computing $\kappa_G(s,t)$ that is based on the following construction: For the given graph $G = (V, E)$ having n vertices and m edges we derive a directed graph $\bar{G} = (\bar{V}, \bar{E})$ with $|\bar{V}| = 2n$ and $|\bar{E}| = 2m+n$ by replacing each vertex $v \in V$ with two vertices $v', v'' \in \bar{V}$ connected by an (internal) edge $e_v = (v', v'') \in \bar{E}$. Every edge $e = (u, v) \in E$ is replaced by two (external) edges $e' = (u'', v'), e'' = (v'', u') \in \bar{E}$, see Figure 7.9.

$\kappa(s,t)$ is now computed as the maximum flow in $\bar{G}$ from source s'' to the target t' with unit capacities for all edges[2]. For a proof of correctness see [187]. For each pair $v', v'' \in \bar{V}$ representing a vertex $v \in V$ the internal edge (v', v'') is the only edge that emanates from v' and the only edge entering v'', thus the network $\bar{G}$ is of type 2. According to Lemma 7.6.2 the computation of the maximum flow resp. the local vertex-connectivity has time complexity $\mathcal{O}(\sqrt{n}m)$.

A trivial algorithm for computing $\kappa(G)$ could determine the minimum for the local connectivity of all pairs of vertices. Since $\kappa_G(s,t) = n-1$ for all pairs (s,t) that are directly connected by an edge, this algorithm would make $\frac{n(n-1)}{2} - m$ calls to the flow-based subroutine. We will see that we can do much better.

If we consider a minimum vertex separator $S \subset V$ that separates a 'left' vertex subset $L \subset V$ from a 'right' subset $R \subset V$, we could compute $\kappa(G)$ by fixing one vertex s in either subset L or R and computing the local connectivities $\kappa_G(s,t)$ for all vertices $t \in V \setminus \{s\}$ one of which must lie on the other side of the vertex cut. The problem is: how to select a vertex s such that s does not belong to every minimum vertex separator? Since $\kappa(G) \leq \delta(G)$ (see Theorem 7.1.1), we could try $\delta(G)+1$ vertices for s, one of which must not be part of all minimum vertex cuts. This would result in an algorithm of complexity $\mathcal{O}((\delta+1) \cdot n \cdot \sqrt{n}m)) = \mathcal{O}(\delta n^{3/2} m)$

Even and Tarjan [188] proposed Algorithm 13 that stops computing the local connectivities if the size of the current minimum cut falls below the number of examined vertices.

The resulting algorithm examines not more than $\kappa + 1$ vertices in the loop for variable i. Each vertex has at least $\delta(G)$ neighbors, thus at most $\mathcal{O}((n - \delta - 1)(\kappa + 1))$ calls to the maximum flow subroutine are carried out. Since $\kappa(G) \leq 2m/n$ (see Theorem 7.1.8), the minimum capacity is found not later than in call $2m/n + 1$. As a result, the overall time complexity is $\mathcal{O}(\sqrt{n}m^2)$.

[2] Firstly, Even used $c(e_v) = 1$, $c(e') = c(e'') = \infty$ which leads to the same results.

Algorithm 13: Vertex-connectivity computation by Even & Tarjan

Input : An (undirected) graph $G = (V, E)$
Output: $\kappa(G)$

$\kappa_{\min} \leftarrow n - 1$
$i \leftarrow 1$
while $i \leq \kappa_{\min}$ **do**
  for $j \leftarrow i + 1$ ***to*** n **do**
    if $i > \kappa_{\min}$ **then**
      break
    else if $\{v_i, v_j\} \notin E$ **then**
      compute $\kappa_G(v_i, v_j)$ using the MaxFlow algorithm
      $\kappa_{\min} \leftarrow \min\{\kappa_{\min}, \kappa_G(v_i, v_j)\}$
return $\kappa_{\min}$

Esfahanian and Hakimi [183] further improved the algorithm by the following observation:

Lemma 7.6.3. *If a vertex v belongs to all minimum vertex-separators then there are for each minimum vertex-cut S two vertices $l \in L_S$ and $r \in R_S$ that are adjacent to v.*

Proof. Assume v takes part in all minimum vertex-cuts of G. Consider the partition of the vertex set V induced by a minimum vertex-cut S with a component L (the 'left' side) of the remaining graph and the respective 'right' side R. Each side must contain at least one of v's neighbors, because otherwise v would not be necessary to break the graph into parts. Actually each side having more than one vertex must contain 2 neighbors since otherwise replacing v by the only neighbor would be a minimum cut without v, in contrast to the assumption. □

These considerations suggest Algorithm 14. The first loop makes $n - \delta - 1$ calls to the MaxFlow procedure, the second requires $\kappa(2\delta - \kappa - 3)/2$ calls. The overall complexity is thus $n - \delta - 1 + \kappa(2\delta - \kappa - 3)/2$ calls of the maximum flow algorithm.

7.6.2 Edge-Connectivity Algorithms

Similar to the computation of the vertex-connectivity, the calculation of the edge-connectivity is based on a maximum-flow algorithm that solves the local edge-connectivity problem, i.e. the computation of $\lambda_G(s, t)$. Simply replace all undirected edges by pairs of antiparallel directed edges with capacity 1 and compute the maximum flow from the source s to the sink t. Since the resulting network is of type 1, the computation is, due to Lemma 7.6.2, of complexity $\mathcal{O}(\min\{m^{3/2}, n^{2/3}m\})$.

A trivial algorithm for computing $\lambda(G)$ could simply calculate the minimum of the local edge-connectivities for all vertex pairs. This algorithm would thus make $n(n-1)/2$ calls to the MaxFlow subroutine. We can easily improve the

Algorithm 14: Vertex-connectivity computation by Esfahanian & Hakimi

```
Input  : An (undirected) graph G = (V, E)
Output: κ(G)

κ_min ← n − 1
Choose v ∈ V having minimum degree, d(v) = δ(G)
Denote the neighbors N(v) by v_1, v_2, ..., v_δ

foreach non-neighbor w ∈ V \ (N(v) ∪ {v}) do
    compute κ_G(v, w) using the MaxFlow algorithm
    κ_min ← min{κ_min, κ_G(v, w)}

i ← 1
while i ≤ κ_min do
    for j ← i + 1 to δ − 1 do
        if i ≥ δ − 2 or i ≥ κ_min then
            return κ_min

        else if {v, w} ∉ E then
            compute κ_G(v_i, v_j) using the MaxFlow algorithm
            κ_min ← min{κ_min, κ_G(v_i, v_j)}
    i ← i + 1
return κ_min
```

complexity of the algorithm if we consider only the local connectivities $\lambda_G(s,t)$ for a single (fixed) vertex s and all other vertices t. Since one of the vertices $t \in V \setminus \{s\}$ must be separated from s by an arbitrary minimum edge-cut, $\lambda(G)$ equals the minimum of all these values. The number of MaxFlow calls is thereby reduced to $n-1$. The overall time complexity is thus $\mathcal{O}(nm \cdot \min\{n^{2/3}, m^{1/2}\})$ (see also [188]). The aforementioned algorithm also works if the whole vertex set is replaced by a subset that contains two vertices that are separated by some minimum edge-cut. Consequently, the next algorithms try to reduce the size of this vertex set (which is called a λ-*covering*). They utilize the following lemma. Let S be a minimum edge-cut of a graph $G = (V, E)$ and let $L, R \subset V$ be a partition of the vertex set such that L and R are separated by S.

Lemma 7.6.4. *If $\lambda(G) < \delta(G)$ then each component of $G - S$ consists of more than $\delta(G)$ vertices, i.e. $|L| > \delta(G)$ and $|R| > \delta(G)$.*

Table 7.4. The history of edge-connectivity algorithms

Year	Author(s)	MaxFlow calls	Check k-EC Compute λ
1975	Even, Tarjan [188]		
		$n-1$	$\mathcal{O}(nm \cdot \min\{n^{2/3}, m^{1/2}\})$
1984	Esfahanian, Hakimi [183]		
		$< n/2$	$\mathcal{O}(\lambda nm)$
1987	Matula [413]		$\mathcal{O}(kn^2)$
			$\mathcal{O}(\lambda n^2)$

Proof. Let the elements of L be denoted by $\{l_1, l_2, \ldots, l_k\}$ and denote the induced edges by $E[L] = E(G[L])$.

$$\begin{aligned}\delta(G) \cdot k &\leq \sum_{i=1}^{k} d_G(l_i) \\ &\leq 2 \cdot |E[L]| + |S| \\ &\leq 2 \cdot \frac{k(k-1)}{2} + |S| \\ &< k(k-1) + \delta(G)\end{aligned}$$

From $\delta(G) \cdot (k-1) < k(k-1)$ we conclude $|L| = k > 1$ and $|L| = k > \delta(G)$ (as well as $|R| > \delta(G)$). □

Corollary 7.6.5. *If $\lambda(G) < \delta(G)$ then each component of $G - S$ contains a vertex that is not incident to any of the edges in S.*

Lemma 7.6.6. *Assume again that $\lambda(G) < \delta(G)$. If T is a spanning tree of G then all components of $G - S$ contain at least one vertex that is not a leaf of T (i.e. the non-leaf vertices of T form a λ-covering).*

Proof. Assume the converse, that is all vertices in L are leaves of T. Thus no edge of T has both ends in L, i.e. $|L| = |S|$. Lemma 7.6.4 immediately implies that $\lambda(G) = |S| = |L| > \delta(G)$, a contradiction to the assumption. □

Lemma 7.6.6 suggests an algorithm that first computes a spanning tree of the given graph, then selects an arbitrary inner vertex v of the tree and computes the local connectivity $\lambda(v, w)$ to each other non-leaf vertex w. The minimum of these values together with $\delta(G)$ yields exactly the edge connectivity $\lambda(G)$. This algorithm would profit from a larger number of leaves in T but, unfortunately, finding a spanning tree with maximum number of leaves is $\mathcal{NP}$-hard.Esfahanian

Algorithm 15: Spanning tree computation by Esfahanian & Hakimi

Input : An (undirected) graph $G = (V, E)$
Output: Spanning Tree T with a leaf and an inner vertex in L and R, resp.

Choose $v \in V$
$T \leftarrow$ all edges incident at v
while $|E(T)| < n - 1$ **do**
  Select a leaf w in T such that for all leaves r in T:
  $|N(w) \cap (V - V(T))| \geq |N(r) \cap (V - V(T))|$
  $T \leftarrow T \cup G[w \cup \{N(w) \cap (V - V(T))\}]$
return T

and Hakimi [183] proposed an algorithm for computing a spanning tree T of G such that both, L and R of some minimum edge separator contain at least one leaf of T, and due to Lemma 7.6.6 at least one inner vertex (see Algorithm 15).The edge-connectivity of the graph is then computed by Algorithm 16. Since P is

Algorithm 16: Edge-connectivity computation by Esfahanian & Hakimi

Input : An (undirected) graph $G = (V, E)$
Output: $\lambda(G)$

Construct a spanning tree T using Algorithm 15
Let P denote the smaller of the two sets, either the leaves or the inner nodes of T
Select a vertex $u \in P$
$c \leftarrow \min\{\lambda_G(u, v) : v \in P \setminus \{u\}\}$
$\lambda \leftarrow \min(\delta(G), c)$
return λ

chosen to be the smaller of both sets, leaves and non-leaves, the algorithm requires at most $n/2$ calls to the computation of a local connectivity, which yields an overall complexity of $\mathcal{O}(\lambda mn)$.

This could be improved by Matula [413], who made use of the following lemma.

Lemma 7.6.7. *In case $\lambda(G) < \delta(G)$, each dominating set of G is also a λ-covering of G.*

Similar to the case of the spanning tree, the edge-connectivity can now be computed by choosing a dominating set D of G, selecting an arbitrary vertex $u \in D$, and calculating the local edge-connectivities between u and all other vertices in D. The minimum of all values together with the minimum degree $\delta(G)$ gives the result. While finding a dominating set of minimum cardinality is $\mathcal{NP}$-hard in general, the connectivity algorithm can be shown to run in time $\mathcal{O}(nm)$ if the dominating set is chosen according to Algorithm 17.

Algorithm 17: Dominating set computation by Matula

Input : An (undirected) graph $G = (V, E)$
Output: A dominating set D

Choose $v \in V$
$D \leftarrow \{v\}$
while $V \setminus (D \cup N(D)) \neq \emptyset$ **do**
    Select a vertex $w \in V \setminus (D \cup N(D))$
    $D \leftarrow D \cup \{w\}$
return D

7.7 Non-flow-based Algorithms

We consider now connectivity algorithms that are not based on network flow techniques.

7.7.1 The Minimum Cut Algorithm of Stoer and Wagner

In 1994 an algorithm for computing a minimum capacity cut of an edge-weighted graph was published by Stoer and Wagner [536]. It was unusual not only due to the fact that it did not use any maximum flow technique as a subroutine. Somewhat surprisingly, the algorithm is very simple in contrast to all other algorithms (flow-based and non-flow-based) that were published so far. In principle, each phase of the algorithm is very similar to Prim's minimum spanning tree algorithm and Dijkstra's shortest path computation, which leads to an equivalent running time of $\mathcal{O}(m + n \log n)$ per phase and overall time complexity of $\mathcal{O}(nm + n^2 \log n)$.

Algorithm 18: Minimum capacity cut computation by Stoer & Wagner

```
Input  : An undirected graph G = (V, E)
Output: A minimum cut C_min corresponding to λ(G)

Choose an arbitrary start vertex a
C_min ← undefined
V' ← V
while |V'| > 1 do
    A ← {a}
    while A ≠ V' do
        Add to A the most tightly connected vertex
        Adjust the capacities between A and the vertices in V' \ A
    C := cut of V' that separates the vertex added last to A from the rest of
    the graph
    if C_min = undefined or w(C) < w(C_min) then
        C_min ← C
    Merge the two vertices that were added last to A
return C_min
```

After choosing an arbitrary start vertex a, the algorithm maintains a vertex subset A that is initialized with the start vertex and that grows by repeatedly adding a vertex $v \notin A$ that has a maximum sum of weights for its connections to vertices in A. If all vertices have been added to A, the last two vertices s and t are merged into one. While edges between s and t are simply deleted by the contraction, all edges from s and t to another vertex are replaced by an edge weighted with the sum of the old weights. The cut that separates the vertex added last from the rest of the graph is called the *cut-of-the-phase*.

Lemma 7.7.1. *The cut-of-the-phase is a minimum s-t-cut in the current (modified) graph, where s and t are the two vertices added last to A in the phase.*

Proof. Consider an arbitrary s-t-cut C for the last two vertices. A vertex $v \neq a$ is called *active* if v and its immediate predecessor with respect to the addition to A reside in different parts of C. Let A_v be the set of vertices that are in A just before v is added and let $w(S, v)$ for a vertex set S denote the capacity sum of all edges between v and the vertices in S.

The proof shows, by induction on the active vertices, that for each active vertex v the adjacency to the vertices added before (A_v) does not exceed the weight of the cut of $A_v \cup \{v\}$ induced by C (denoted by C_v). Thus it is to prove that

$$w(A_v, v) \leq w(C_v)$$

For the base case, the inequality is satisfied since both values are equal for the first active vertex. Assuming now that the proposition is true for all active vertices up to active vertex v, the value for the next active vertex u can be written as

$$\begin{array}{lcll} w(A_u, u) & = & w(A_v, u) + w(A_u \setminus A_v, u) & \\ & \leq & w(A_v, v) + w(A_u \setminus A_v, u) & (w(A_v, u) \leq w(A_v, v)) \\ & \leq & w(C_v) + w(A_u \setminus A_v, u) & \text{(by induction assumption)} \\ & \leq & w(C_u) & \end{array}$$

The last line follows because all edges between $A_u \setminus A_v$ and u contribute their weight to $w(C_u)$ but not to $w(C_v)$.

Since t is separated by C from its immediate predecessor s, it is always an active vertex; thus the conclusion $w(A_t, t) \leq w(C_t)$ completes the proof. □

Theorem 7.7.2. *A cut-of-the-phase having minimum weight among all cuts-of-the-phase is a minimum capacity cut of the original graph.*

Proof. For the case where the graph consists of only 2 vertices, the proof is trivial. Now assume $|V| > 2$. The following two cases can be distinguished:

1. Either the graph has a minimum capacity cut that is also a minimum s-t-cut (where s and t are the vertices added last in the first phase), then, according to Lemma 7.7.1, we conclude that this cut is a minimum capacity cut of the original graph.
2. Otherwise the graph has a minimum cut where s and t are on the same side. Therefore the minimum capacity cut is not affected by merging the vertices s and t.

Thus, by induction on the number of vertices, the minimum capacity cut of the graph is the cut-of-the-phase having minimum weight. □

7.7.2 Randomized Algorithms

In 1982, Becker et al. [53] proposed a probabilistic variant of the Even/Tarjan vertex connectivity algorithm [188]. It computes the vertex connectivity of an undirected graph G with error probability at most ε in expected time $\mathcal{O}((-\log \varepsilon)n^{3/2}m)$ provided that $m \leq \frac{1}{2}dn^2$ for some constant $d < 1$. This improved the computation of κ for sparse graphs.

A few years later, Linial, Lovasz and Wigderson provided probabilistic algorithms [392, 393] that were based on a geometric, algebraic and physical interpretation of graph connectivity. As a generalization of the notion of s-t-numbering, they showed that a graph G is k-connected if and only if it has a certain non-degenerate convex embedding in $\mathbb{R}^{k-1}$, i.e., specifying any k vertices of G, the

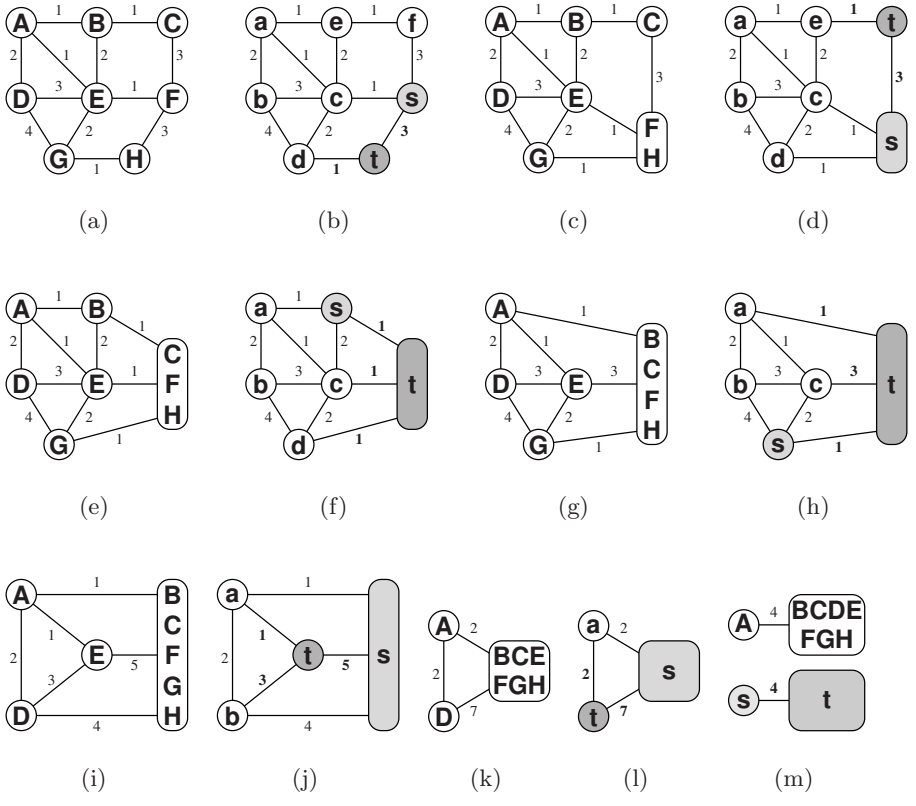


Fig. 7.10. Example for the Stoer/Wagner algorithm. Upper case letters are vertex names, lower case letters show the order of addition to the set S. The minimum cut $\{ABDEG\} \mid \{CFH\}$ has capacity 3 and is found in Part 7.10(f) (third phase)

vertices of G can be represented by points of $\mathbb{R}^{k-1}$ such that no k are in a hyperplane and each vertex is in the convex hull of its neighbors, except for the k specified vertices. As a result, they proposed a Monte-Carlo algorithm running in time $\mathcal{O}(n^{2.5} + n\kappa^{2.5})$ (that errs with probability less than $1/n$) and a Las Vegas algorithm with expected runtime of $\mathcal{O}(n^{2.5} + n\kappa^{3.5})$.

A subsequent work of Cheriyan and Reif [120] generalized this approach to directed graphs, which yielded a Monte Carlo algorithm with running time $\mathcal{O}((M(n) + nM(k)) \cdot \log n)$ and error probability $< 1/n$, and a Las Vegas algorithm with expected time $\mathcal{O}((M(n) + nM(k)) \cdot k)$, where $M(n)$ denotes the complexity for the multiplication of $n \times n$ matrices.

Henzinger, Rao and Gabow [298] further improved the complexities by giving an algorithm that computes the vertex connectivity with error probability at most $1/2$ in (worst-case) time $\mathcal{O}(nm)$ for digraphs and $\mathcal{O}(\kappa n^2)$ for undirected

graphs. For weighted graphs they proposed a Monte Carlo algorithm that has error probability 1/2 and expected running time $\mathcal{O}(nm \log(n^2/m))$.

7.8 Basic Algorithms for Components

Super-linear algorithms for the computation of the blocks and the cut-vertices as well as for the computation of the strongly connected components of a graph were proposed in [470] and [386, 484, 485, 435], respectively. Later on, linear time algorithms were published by Hopcroft and Tarjan [311, 542].

7.8.1 Biconnected Components

A problem that arises from the question which nodes of a network always remain connected in case one arbitrary node drops out is the computation of the *biconnected (or non-separable) components* of a graph, also called *blocks*.

Let us consider a depth-first search in an undirected and connected graph $G = (V, E)$ where we label the traversed vertices with consecutive numbers from 1 to $n = |V|$ using a pre-order numbering `num`. We observe that we inspect two kinds of edges: the ones that lead to unlabeled vertices become *tree edges*, and the ones that lead to vertices that were already discovered and labeled in a former step we call *backward edges*.

For each vertex v we keep the smallest label of any vertex that is reachable via arbitrary tree edges followed by not more than one backward edge, i.e. the smallest number of any vertex that lies on some cycle with v. Whenever a new vertex is discovered by the DFS, the `low`-entry of that vertex is initialized by its own number.

If we return from a descent to a child w – i.e. from a tree edge (v, w) –, we update `low`[v] by keeping the minimum of the child's entry `low`[w] and the current value `low`[v].

If we discover a backward edge (v, w), we update `low`[v] to be the minimum of its old value and the label of w.

To detect the cut-vertices of the graph we can now utilize the following lemma:

Lemma 7.8.1. *We follow the method described above for computing the values of* `low` *and* `num` *during a DFS traversal of the graph G. A vertex v is a cut-vertex if and only if one of the following conditions holds:*

1. *if v is the root of the DFS tree and is incident to at least 2 DFS tree edges,*
2. *if v is not the root, but there is a child w of v such that* `low`$[w] \geq$ `num`$[v]$.

Proof. 1. Assume that v is the root of the DFS tree.
→ If v is incident to more than one tree edge, the children would be disconnected by removing vertex v from G.

← If v is a cut-vertex then there are vertices $x, y \in V$ that are disconnected by removing v, i.e. v is on every path connecting x and y. W.l.o.g. assume that the DFS discovers x before y. y can only be discovered after the descent to x returned to v, thus we conclude that v has at least two children in the DFS tree.

2. Assume now that v is not the root of the DFS tree.
   → If there is a child w of v such that $\texttt{low}[w] \geq \texttt{num}[v]$ this means that there is only one path connecting this successor w with all ancestors of v. Thus v is a cut-vertex.
   ← If v is a cut-vertex, there are vertices $x, y \in V$ such that v is on every path connecting x and y. If all children of v had an indirect connection (via arbitrary tree edges followed by one backward edge) to any ancestor of v the remaining graph would be connected. Therefore one of the children must have $\texttt{low}[w] \geq \texttt{num}[v]$.

This concludes the proof. □

To find the biconnected components, i.e. the partition of the edges, we put every new edge on a stack. Whenever the condition $\texttt{low}[w] \geq \texttt{num}[v]$ holds after returning from a recursive call for a child w of v, the edges on top of stack including edge (v, w) form the next block (and are therefore removed from the stack).

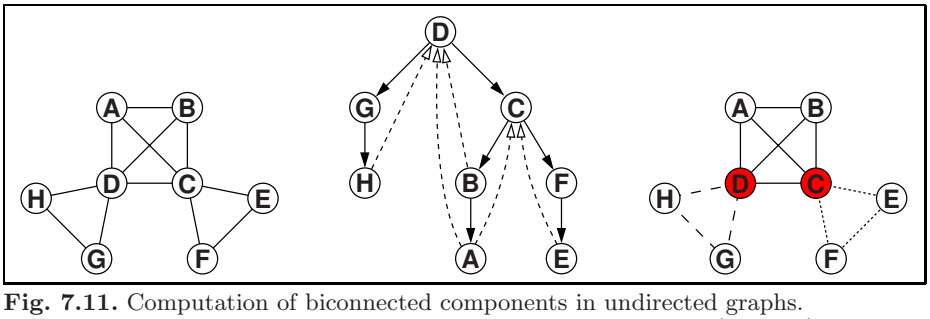


Fig. 7.11. Computation of biconnected components in undirected graphs. Left: the undirected input graph. Middle: dfs tree with forward (straight) and backward (dashed) edges. Right: the blocks and articulation nodes of the graph.

7.8.2 Strongly Connected Components

We now consider the computation of the strong components, i.e. the maximal strongly connected subgraphs in directed graphs (see Section 2.2.1). Analogously to the computation of biconnected components in undirected graphs, we use a modified depth-first search that labels the vertices by consecutive numbers from 1 to n. In case the traversal ends without having discovered all vertices we have to restart the DFS at a vertex that has not been labeled so far. The result is a spanning forest F.

The edges $e = (v, w)$ that are inspected during the DFS traversal are divided into the following categories:

1. All edges that lead to unlabeled vertices are called *tree edges* (they belong to the trees of the DFS forest).
2. The edges that point to a vertex w that was already labeled in a former step fall into the following classes:
   a) If $\texttt{num}[w] > \texttt{num}[v]$ we call e a *forward edge*.
   b) Otherwise, if w is an ancestor of v in the same DFS tree we call e a *backward edge*.
   c) Otherwise e is called a *cross edge* (because it points from one subtree to another).

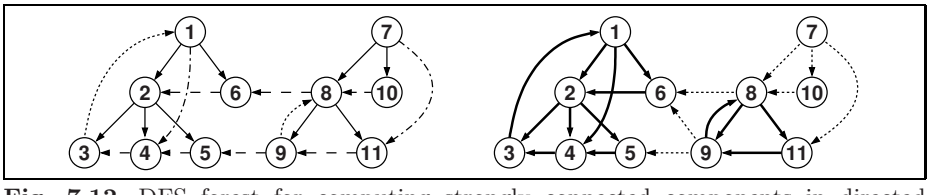


Fig. 7.12. DFS forest for computing strongly connected components in directed graphs: tree, forward, backward, and cross edges

An example is shown in Figure 7.12.

Two vertices v, w are in the same strong component if and only if there exist directed paths from v to w and from w to v. This induces an equivalence relation as well as a partition of the vertex set (in contrast to biconnected components where the edge set is partitioned while vertices may belong to more than one component).

During the DFS traversal we want to detect the roots of the strong components, i.e. in each component the vertex with smallest DFS label. As in the case of the biconnected components we must decide for each descendant w of a vertex v whether there is also a directed path that leads back from w to v. Now we define `lowlink`[v] to be the smallest label of any vertex in the same strong component that can be reached via arbitrarily many tree arcs followed by at most one backward or cross edge.

Lemma 7.8.2. *A vertex v is the root of a strong component if and only if both of the following conditions are met:*

1. *There is no backward edge from v or one of its descendants to an ancestor of v.*
2. *There is no cross edge (v, w) from v or one of its descendants to a vertex w such that the root of w's strong component is an ancestor of v.*

This is equivalent with the decision whether $\texttt{lowlink}[v] = \texttt{num}[v]$.

Proof. $\rightarrow$ Assume conversely that the condition holds but u is the root of v's strong component with $u \neq v$. There must exist a directed path from v to u. The first edge of this path that points to a vertex w that is not a descendant of v in the DFS tree is a back or a cross edge. This implies $\texttt{lowlink}[v] \leq \texttt{num}[w] < \texttt{num}[v]$, since the highest numbered common ancestor of v and w is also in this strong component.

$\leftarrow$ If v is the root of some strong component in the actual spanning forest, we may conclude that $\texttt{lowlink}[v] = \texttt{num}[v]$. Assuming the opposite (i.e. $\texttt{lowlink}[v] < \texttt{num}[v]$), some proper ancestor of v would belong to the same strong component. Thus v would not be the root of the SCC.

This concludes the proof. □

If we put all discovered vertices on a stack during the DFS traversal (similar to the stack of edges in the computation of the biconnected components) the lemma allows us to 'cut out' the strongly connected components of the graph.

It is apparent that the above algorithms share their similarity due to the fact that they are based on the detection of cycles in the graph. If arbitrary instead of simple cycles (for biconnected components) are considered, this approach yields a similar third algorithm that computes the bridge- (or 2-edge-) connected components (published by Tarjan [544]).

7.8.3 Triconnectivity

First results on graph triconnectivity were provided by Mac Lane [403] and Tutte [555, 556]. In the sixties, Hopcroft and Tarjan published a linear time algorithm for dividing a graph into its triconnected components that was based on depth-first search [309, 310, 312]. Miller and Ramachandran [422] provided another algorithm based on a method for finding open ear decompositions together with an efficient parallel implementation. It turned out that the early Hopcroft/Tarjan algorithm was incorrect, which was then modified by Gutwenger and Mutzel [267]. They modified the faulty parts to yield a correct linear time implementation of SPQR-trees. We now briefly review their algorithm.

Definition 7.8.3. *Let $G = (V, E)$ be a biconnected (multi-) graph. Two vertices $a, b \in V$ are called a* separation pair *of G if the induced subgraph on the vertices $V \setminus \{a, b\}$ is not connected.*

The pair (a, b) partitions the edges of G into equivalence classes $E_1, \ldots, E_k$ (*separation classes*), s.t. two edges belong to the same class exactly if both lie on some path p that contains neither a nor b as an inner vertex, i.e. if it contains a or b it is an end vertex of p. The pair (a, b) is a separation pair if there are at least two separation classes, except for the following special cases: there are exactly two separation classes, and one of them consists of a single edge, or if there are exactly three separation classes that all consist of a single edge. The graph G is triconnected if it contains no separation pair.

Definition 7.8.4. *Let (a, b) be a separation pair of a biconnected multigraph G and let the separation classes $E_{1..k}$ be divided into two groups $E' = \bigcup_{i=1}^{l} E_i$ and $E'' = \bigcup_{i=l+1}^{k} E_i$, s.t. each group contains at least two edges. The two graphs $G' = (V(E' \cup e), E' \cup e)$ and $G'' = (V(E'' \cup e), E'' \cup e)$ that result from dividing the graph according to the partition $[E', E'']$ and adding the new* virtual edge $e = (a, b)$ *to each part are called* split graphs *of G (and they are again biconnected). If the split operation is applied recursively to the split graphs, this yields the (not necessarily unique)* split components *of G.*

Every edge in E is contained in exactly one, and each virtual edge in exactly two split components.

Lemma 7.8.5. *Let $G = (V, E)$ be a biconnected multigraph with $|E| \geq 3$. Then the total number of edges contained in all split components is bounded by $3|E|-6$.*

Proof. Induction on the number of edges of G: If $|E| = 3$, G cannot be split and the lemma is true. Assume now, the lemma is true for graphs having at most $m - 1$ edges. If the graph has m edges, the lemma is obviously true if G cannot be split. Otherwise G can be split into two graphs having $k + 1$ and $m - k + 1$ edges with $2 \leq k \leq m - 2$. By the assumption, the total number of edges is bounded by $3(k + 1) - 6 + 3(m - k + 1) - 6 = 3m - 6$. Thus, by induction on the number of edges, the proof is complete. □

There are split components of three types: triple bonds (three edges between two vertices), triangles (cycles of length 3), and triconnected simple graphs. We now introduce the reverse of the split operation: the *merge graph* of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, both containing the same virtual edge e, is defined as $G = (V_1 \cup V_2, (E_1 \cup E_2) \setminus \{e\})$. The *triconnected components* of a graph are obtained from its split components by merging the triple bonds as much as possible to multiple bonds and by merging the triangles as much as possible to form polygons. Mac Lane [403] showed that, regardless of the (possibly not unique) splitting and merging, we get the same triconnected components.

Lemma 7.8.6. *The triconnected components of a (multi)graph are unique.*

We now turn to the definition of SPQR-trees, which were initially defined for planar [143], later also for general graphs [144]. A *split pair* of a biconnected graph G is either a separation pair or a pair of adjacent vertices. A *split component* of a split pair $\{u, v\}$ is either an (u, v)-edge or an inclusion-maximal subgraph of G, were $\{u, v\}$ is not a split pair. A split pair $\{u, v\}$ of G is called a *maximal split pair* with respect to a split pair $\{s, t\}$ of G if for any other split pair $\{u', v'\}$, the vertices u, v, s, and t are in the same split component.

Definition 7.8.7. *Let $e = (s, t)$ be an edge of G. The* SPQR-tree *$\mathcal{T}$ of G with respect to this* reference edge *is a rooted ordered tree constructed from four different types of nodes (S,P,Q,R), each containing an associated biconnected multigraph (called the* skeleton*). $\mathcal{T}$ is recursively defined as follows:*

(Q) Trivial Case: If G consists of exactly two parallel s-t-edges, then $\mathcal{T}$ is a single Q-node with skeleton G.

(P) Parallel Case: If the split pair $\{s,t\}$ *has more than two split components* $G_{1..k}$*, the root of* $\mathcal{T}$ *is a P-node with a skeleton consisting of* k *parallel* s*-*t*-edges* $e_{1..k}$ *with* $e_1 = e$*.*

(S) Series Case: If the split pair $\{s,t\}$ *has exactly two split components, one of them is* e*; the other is denoted by* G'*. If* G' *has cut-vertices* $c_{1..k-1} (k \geq 2)$ *that partition* G *into blocks* $G_{1..k}$ *(ordered from* s *to* t*), the root of* $\mathcal{T}$ *is an S-node, whose skeleton is the cycle consisting of the edges* $e_{0..k}$*, where* $e_0 = e$ *and* $e_i = (c_{i-1}, c_i)$ *with* $i = 1..k$*,* $c_0 = s$ *and* $c_k = t$*.*

(R) Rigid Case: In all other cases let $\{s_1,t_1\},..,\{s_k,t_k\}$ *be the maximal split pairs of* G *with respect to* $\{s,t\}$*. Further let* G_i *for* $i = 1,..,k$ *denote the union of all split components of* $\{s_i,t_i\}$ *except the one containing* e*. The root of* $\mathcal{T}$ *is an R-node, where the skeleton is created from* G *by replacing each subgraph* G_i *with the edge* $e_i = (s_i,t_i)$*.*

For the non-trivial cases, the children $\mu_{1..k}$ *of the node are the roots of the SPQR-trees of* $G_i \cup e_i$ *with respect to* e_i*. The vertices incident with each edge* e_i *are the* poles *of the node* μ_i*, the virtual edge of node* μ_i *is the edge* e_i *of the node's skeleton. The SPQR-tree* $\mathcal{T}$ *is completed by adding a Q-node as the parent of the node, and thus the new root (that represents the reference edge* e*).*

Each edge in G corresponds with a Q-node of $\mathcal{T}$, and each edge e_i in the skeleton of a node corresponds with its child μ_i. $\mathcal{T}$ can be rooted at an arbitrary Q-node, which results in an SPQR-tree with respect to its corresponding edge.

Theorem 7.8.8. *Let* G *be a biconnected multigraph with SPQR-tree* $\mathcal{T}$*.*

1. *The skeleton graphs of* $\mathcal{T}$ *are the triconnected components of* G*. P-nodes correspond to bonds, S-nodes to polygons, and R-nodes to triconnected simple graphs.*
2. *There is an edge between two nodes* $\mu, \nu \in \mathcal{T}$ *if and only if the two corresponding triconnected components share a common virtual edge.*
3. *The size of* $\mathcal{T}$*, including all skeleton graphs, is linear in the size of* G*.*

For a sketch of the proof, see [267].

We consider now the computation of SPQR-trees for a biconnected multigraph G (without self-loops) and a reference edge e_r. We assume a labeling of the vertices by unique indices from 1 to $|V|$. As a preprocessing step, all edges are reordered (using bucket sort), first according to the incident vertex with the lower index, and then according to the incident vertex with higher index, such that multiple edges between the same pair of vertices are arranged successively. In a second step, all such bundles of multiple edges are replaced by a new virtual edge. In this way a set of multiple bonds $C_1,..,C_k$ is created together with a simple graph G'.

In the second step, the split components $C_{k+1},..,C_m$ of G' are computed using a dfs-based algorithm. In this context, we need the following definition:

Definition 7.8.9. *A* palm tree *P is a directed multigraph that consists of a set of* tree arcs *$v \to w$ and a set of* fronds *$v \hookrightarrow w$, such that the tree arcs form*

a directed spanning tree of P (that is the root has no incoming edges, all other vertices have exactly one parent), and if $v \hookrightarrow w$ is a frond, then there is a directed path from w to v.

Suppose now, P is a palm tree for the underlying simple biconnected graph $G' = (V, E')$ (with vertices labeled $1, .., |V|$). The computation of the separation pairs relies on the definition of the following variables:

$$\text{lowpt1}(v) = \min\Big(\{v\} \cup \{w | v \overset{*}{\rightarrow\!\!\!\!\hookrightarrow} w\}\Big)$$
$$\text{lowpt2}(v) = \min\Big(\{v\} \cup \Big(\{w | v \overset{*}{\rightarrow\!\!\!\!\hookrightarrow} w\} \setminus \{\text{lowpt1}(v)\}\Big)\Big)$$

These are the two vertices with minimum label, that are reachable from v by traversing an arbitrary number (including zero) of tree arcs followed by exactly one frond of P (or v itself, if no such option exists).

Let $\text{Adj}(v)$ denote the ordered adjacency list of vertex v, and let $D(v)$ be the set of descendants of v (that is the set of vertices that are reachable via zero or more directed tree arcs). Hopcroft and Tarjan [310] showed a simple method for computing an *acceptable adjacency structure*, that is, an order of the adjacency lists, which meets the following conditions:

1. The root of P is the vertex labeled with 1.
2. If $w_1, .., w_n$ are the children of vertex v in P according to the ordering in $\text{Adj}(v)$, then $w_i = v + |D(w_{i+1} \cup .. \cup D(w_n)| + 1$,
3. The edges in $\text{Adj}(v)$ are in ascending order according to $\text{lowpt1}(w)$ for tree edges $v \rightarrow w$, and w for fronds $v \hookrightarrow w$, respectively.
   Let $w_1, .., w_n$ be the children of v with $\text{lowpt1}(w_i)) = u$ ordered according to $\text{Adj}(v)$, and let i_0 be the index such that $\text{lowpt2}(w_i) < v$ for $1 \leq i \leq i_0$ and $\text{lowpt2}(w_j) \geq v$ for $i_0 < j \leq n$. Every frond $v \hookrightarrow w \in E'$ resides between $v \rightarrow w_{i_0}$ and $v \rightarrow w_{i_0+1}$ in $\text{Adj}(v)$.

An adequate rearrangement of the adjacency structure can be done in linear time if a bucket sort with $3|V| + 2$ buckets is applied to the following sorting function (confer [310, 267]), that maps the edges to numbers from 1 to $3|V| + 2$:

$$\phi(e) = \begin{cases} 3\text{lowpt1}(w) & \text{if } e = v \rightarrow w \text{ and } \text{lowpt2}(w) < v \\ 3w + 1 & \text{if } e = v \hookrightarrow w \\ 3\text{lowpt1}(w) + 2 & \text{if } e = v \rightarrow w \text{ and } \text{lowpt2}(w) \geq v \end{cases}$$

If we perform a depth-first search on G' according to the ordering of the edges in the adjacency list, then this partitions G' into a set of paths, each consisting of zero or more tree arcs followed by a frond, and each path ending at the vertex with lowest possible label. We say that a vertex u_n is a *first descendant* of u_0 if there is a directed path $u_0 \rightarrow \cdots \rightarrow u_n$ and each edge $u_i \rightarrow u_{i+1}$ is the first in $\text{Adj}(u_i)$.

Lemma 7.8.10. *Let P be a palm tree of a biconnected graph $G = (V, E)$ that satisfies the above conditions. Two vertices $a, b \in V$ with $a < b$ form a separation pair $\{a, b\}$ if and only if one of the following conditions is true:*

Type-1 Case *There are distinct vertices $r, s \in V \setminus \{a, b\}$ such that $b \to r$ is a tree edge, lowpt1$(r) = a$, lowpt2$(r) \geq b$, and s is not a descendant of r.*

Type-2 Case *There is a vertex $r \in V \setminus b$ such that $a \to r \overset{*}{\to} b$, b is a first descendant of r (i.e., a, r, b lie on a generated path), $a \neq 1$, every frond $x \hookrightarrow y$ with $r \leq x < b$ satisfies $a \leq y$, and every frond $x \hookrightarrow y$ with $a < y < b$ and $b \to w \overset{*}{\to} x$ has lowpt1$(w) \geq a$.*

Multiple Edge Case *(a, b) is a multiple edge of G and G contains at least four edges.*

For a proof, see [310].

We omit the rather technical details for finding the split components $C_{k+1}, .., C_m$. The main loop of the algorithm computes the triconnected components from the split components $C_1, .., C_m$ by merging two bonds or two polygons that share a common virtual edge (as long as they exist). The resulting time complexity is $\mathcal{O}(|V| + |E|)$. For a detailed description of the algorithm we refer the interested reader to the original papers [309, 310, 312, 267].

7.9 Chapter Notes

In this section, we briefly discuss some further results related to the topic of this chapter.

Strong and biconnected components. For the computation of strongly connected components, there is another linear-time algorithm that was suggested by R. Kosaraju in 1978 (unpublished, see [5, p. 229]) and that was published by Sharir [517].

An algorithm for computing the strongly connected components using a non-dfs traversal (a mixture of dfs and bfs) of the graph was presented by Jiang [331]. This algorithm reduces the number of disk operations in the case where a large graph does not entirely fit into the main memory. Two space-saving versions of Tarjan's strong components algorithm (for the case of graphs that are sparse or have many single-node components) were given by Nuutila and Soisalon-Soininen [454].

One-pass algorithms for biconnected and strong components that do not compute auxiliary quantities based on the dfs tree (e.g., `low` values) were proposed by Gabow [235].

Average connectivity. Only recently, Beineke, Oellermann, and Pippert [56] considered the concept of average connectivity. This measure is defined as the average, over all pairs of vertices $a, b \in V$, of the maximum number of vertex-disjoint paths between a and b, that is, the average local vertex-connectivity. While the conventional notion of connectivity is rather a description of a worst case scenario, the average connectivity might be a better description of the global properties of a graph, with applications in network vulnerability and reliability. Sharp bounds for this measure in terms of the average degree were shown by

Dankelmann and Oellermann [138]. Later on, Henning and Oellermann considered the average connectivity of directed graphs and provided sharp bounds for orientations of graphs [294].

Dynamic Connectivity Problems. Quite a number of publications consider connectivity problems in a dynamical setting, that is, in graphs that are changed by vertex and/or edge insertions and deletions. The special case where only insertions are allowed is called semi-dynamic, partially-dynamic, or incremental. Since there is a vast number of different variants, we provide only the references for further reading: [490, 377, 237, 223, 341, 577, 144, 296, 297, 155, 295, 154, 303].

Directed graphs. As already mentioned, the local connectivity in directed graphs is not symmetric, which is the reason why many algorithms for undirected connectivity problems do not translate to the directed case. Algorithms that compute the edge-connectivity in digraphs were published by Schnorr [503] and by Mansour and Schieber [407]. Another problem of interest is the computation of edge-disjoint branchings, which is discussed in several publications [171, 232, 264, 551, 582].

Other measures. There are some further definitions that might be of interest. Matula [410] defines a *cohesiveness function* for each element of a graph (vertices and edges) to be the maximum edge-connectivity of any subgraph containing that element. Akiyama et al. [13] define the *connectivity contribution* or *cohesiveness* of a vertex v in a graph G as the difference $\kappa(G) - \kappa(G - v)$.

Connectivity problems that aim at dividing the graph into more than two components by removing vertices or edges are considered in conjunction with the following terms: A *shredder* of an undirected graph is a set of vertices whose removal results in at least three components, see for example [121]. The *ℓ-connectivity* of a graph is the minimum number of vertices that must be deleted to produce a graph with at least ℓ components or with fewer than ℓ vertices, see [456, 455]. A similar definition exists for the deletion of edges, namely the *i-th order edge connectivity*, confer [254, 255].

Acknowledgments. The authors thank the anonymous reviewer, the editors, and Frank Schilder for critical assessment of this chapter and valuable suggestions. We thank Professor Ortrud Oellermann for her support.